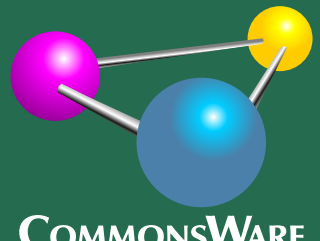
The Busy Coder's Guide to Android Development

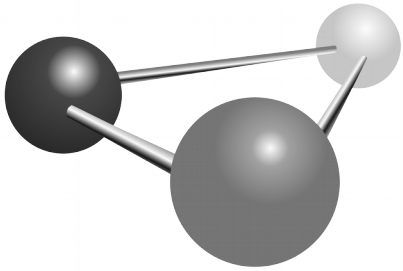
Mark L. Murphy



CommonsWare

The Busy Coder's Guide to Android Development

by Mark L. Murphy



COMMONSWARE

The Busy Coder's Guide to Android Development
by Mark L. Murphy

Copyright © 2008-2018 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
Feburary 2018: Version 8.10 ISBN: 978-0-9816780-0-9

The CommonsWare name and logo, “Busy Coder's Guide”, and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

- Preface
  - Welcome to the Book! xliii
  - The Book's Structure xliii
  - ***The Trails ..... xliv***
  - About the Updates l
  - ***What's New in Version 8.10? ..... l***
  - Warescription l
  - About the APK Edition li
  - Book Bug Bounty lii
  - Source Code and Its License liii
  - ***Creative Commons and the Four-to-Free (42F) Guarantee ..... liv***
  - Acknowledgments liv
- Key Android Concepts
  - Android Applications 1
  - ***Android Devices 6***
  - Don't Be Scared 10
- Choosing Your Development Toolchain
  - Android Studio 11
  - Eclipse 11
  - IntelliJ IDEA 12
  - Command-Line Builds via Gradle 12
  - Yet Other Alternatives 12
  - IDEs... And This Book 13
  - What We Are Not Covering 13
- Tutorial #1 - Installing the Tools
  - But First, Some Notes About Android's Emulator 15
  - Step #1: Checking Your Hardware 16
  - ***Step #2: Setting Up Java and 32-Bit Linux Support 17***
  - ***Step #3: Install Android Studio 17***
  - Step #4: Install the SDKs and Add-Ons 19
  - In Our Next Episode... 27
- Android and Projects
  - ***Projects and Android Studio 29***
  - Starter Project Generators 35
- Tutorial #2 - Creating a Stub Project

- About Our Tutorial Project 37
- About the Rest of the Tutorials 38
- About Our Tools 38
- Step #1: Importing the Project 38
- Step #2: Get Ready for the x86 Emulator 42
- Step #3: Set Up the AVD 43
- Step #4: Set Up the Device 50
- ***Step #5: Running the Project 55***
- In Our Next Episode... 57

• Getting Around Android Studio
- Navigating The Project Explorer 59
- Running Projects 62
- Viewing Output 63
- Accessing Android Tools 64
- Android Studio and Release Channels 66
- Visit the Trails! 67

• Contents of Android Projects
- What You Get, In General 69
- More About the Directory Structure 72
- What You Get Out Of It 74

• Introducing Gradle and the Manifest
- Gradle: The Big Questions 75
- Obtaining Gradle 77
- Versions of Gradle and the Android Gradle Plugin 79
- Gradle Environment Variables 80
- Examining the Gradle Files 80
- Introducing the Manifest 83
- Things In Common Between the Manifest and Gradle 84
- Other Gradle Items of Note 86
- Where's the GUI? 87
- The Rest of the Manifest 87
- Learning More About Gradle 90
- Visit the Trails! 90

• Tutorial #3 - Manifest Changes
- Some Notes About Relative Paths 91
- Step #1: Supporting Screens 91
- Step #2: Blocking Backups 92
- Step #3: Ignoring Lint 92
- In Our Next Episode... 94

• Some Words About Resources
- String Theory 95

- Got the Picture? 101
- Dimensions 105
- The Resource That Shall Not Be Named... Yet 107

- Icons
  - App Icons... And Everything Else 109
  - Creating an App Icon with the Asset Studio 110
  - Creating Other Icons with the Asset Studio 115
- Tutorial #4 - Adjusting Our Resources
  - Step #1: Changing the Name 117
  - Step #2: Changing the Icon 118
  - Step #3: Running the Result 122
  - In Our Next Episode... 123
- The Theory of Widgets
  - What Are Widgets? 125
  - Size, Margins, and Padding 127
  - What Are Containers? 127
  - The Absolute Positioning Anti-Pattern 128
  - The Theme of This Section: Themes 129
- The Android User Interface
  - The Activity 133
  - Dissecting the Activity 134
  - Using XML-Based Layouts 135
- Basic Widgets
  - Common Concepts 139
  - Introducing the Graphical Layout Editor 142
  - And Now, Some Notes About the Book's Sample Projects 151
  - Assigning Labels 152
  - A Commanding Button 157
  - Fleeting Images 160
  - Fields of Green. Or Other Colors. 166
  - More Common Concepts 169
  - Visit the Trails! 172
- Debugging Your App
  - Get Thee To a Stack Trace 174
  - The Case of the Confounding Class Cast 175
  - Point Break 176
- The Classic Container Classes
  - Introducing the Sampler App 177
  - RTL and Your Layouts 178
  - LinearLayout and the Box Model 179
  - All Things Are Relative 203

- Tabula Rasa 214
- Hey, What About ConstraintLayout? 220
- Turning Back to RTL 221

• Other Common Widgets and Containers
- Just a Box to Check 223
- Don't Like Checkboxes? How About Toggles or Switches? 226
- Turn the Radio Up 230
- Scrollwork 233
- Making Progress with ProgressBars 236
- Framing the Scene 237
- Visit the Trails! 238

• Tutorial #5 - Creating a Layout
- Step #1: Creating a New Layout Resource 239
- Step #2: Defining the UI 240
- In Our Next Episode... 245

• GUI Building, Continued
- Making Your Selection 247
- Including Includes 248
- Preview of Coming Attractions 250

• AdapterViews and Adapters
- Adapting to the Circumstances 251
- Lists of Naughty and Nice 253
- Clicks versus Selections 255
- Spin Control 259
- Grid Your Lions (Or Something Like That...) 262
- Fields: Now With 35% Less Typing! 266
- Customizing the Adapter 270
- Visit the Trails! 279

• The WebView Widget
- Role of WebView 281
- Daddy, Where Do WebViews Come From? 282
- Adding the Widget 282
- Loading Content Via a URL 283
- Links and Redirects 285
- Supporting JavaScript 285
- Alternatives for Loading Content 286
- Listening for Events 287
- Addressing the Link/Redirect Behavior 290
- Visit the Trails! 292

• Defining and Using Styles
- Styles: DIY DRY 293

    - Elements of Style 295
    - Themes: Would a Style By Any Other Name... 298
    - What Happens If You Have No Theme 299
    - Android Studio's Theme Editor 299
- Dependencies
    - What's a Dependency? 305
    - Dependency Scopes 306
    - Depending on a Local JAR 307
    - What's an Artifact? 308
    - Artifacts and Repositories 308
    - Major Library Families from Google 308
    - Requesting Dependencies 309
    - The Android Support Library 312
- Tutorial #6 - Adding a Library
    - Step #1: Getting Rid of Existing Cruft 317
    - Step #2: Requesting New Dependencies 318
    - In Our Next Episode... 320
- Introducing ConstraintLayout
    - Why Another Container? 321
    - Comparing with the Classics 323
    - Getting ConstraintLayout 323
    - Using Widgets and Containers from Libraries 325
    - Using a ConstraintLayout 325
    - Converting Existing Layouts 341
    - Visit the Trails! 343
- The Action Bar
    - Bar Hopping 345
    - Yet Another History Lesson 351
    - Your Action Bar Options 352
    - Setting the Target 354
    - Defining the Resource 355
    - Applying the Resource 358
    - Responding to Events 358
    - The Rest of the Sample Activity 359
    - MENU Key, We Hardly Knew Ye 365
    - Action Bars, Live in Living Color! 366
    - Visit the Trails! 376
- Vector Drawables
    - Getting the Artwork 377
    - VectorDrawableCompat 382
    - Other VectorDrawable Backports 390

- Tutorial #7 - Setting Up the Action Bar
  - Step #1: Adding Some Icons 391
  - Step #2: Defining Some Options 393
  - Step #3: Loading and Responding to Our Options 396
  - Step #4: Supporting Older Devices 397
  - Step #5: Trying It Out 398
  - In Our Next Episode... 400
- Android's Process Model
  - When Processes Are Created 401
  - BACK, HOME, and Your Process 402
  - Termination 403
  - Foreground Means "I Love You" 404
  - You and Your Heap 404
- Activities and Their Lifecycles
  - Creating Your Second (and Third and...) Activity 406
  - Warning! Contains Explicit Intents! 410
  - Using Implicit Intents 412
  - Extra! Extra! 418
  - Pondering Parcelable 419
  - Asynchronicity and Results 421
  - Schroedinger's Activity 421
  - Life, Death, and Your Activity 422
  - When Activities Die 424
  - Walking Through the Lifecycle 425
  - Recycling Activities 428
  - Application: Transcending the Activity 429
  - The Case of the Invisible Activity 430
- Tutorial #8 - Setting Up An Activity
  - Step #1: Creating the Stub Activity Class and Manifest Entry 433
  - Step #2: Launching Our Activity 435
  - In Our Next Episode... 436
- The Tactics of Fragments
  - The Six Questions 437
  - Where You Get Your Fragments From 440
  - Your First Fragment 440
  - The Fragment Lifecycle Methods 444
  - Your First Dynamic Fragment 445
  - Fragments and the Action Bar 448
  - Fragments Within Fragments: Just Say "Maybe" 450
  - Fragments and Multiple Activities 450
- Tutorial #9 - Starting Our Fragments

    - Step #1: Create a SimpleContentFragment 453
    - Step #2: Examining SimpleContentFragment 455
    - In Our Next Episode... 456
- Swiping with ViewPager
    - Pieces of a Pager 457
    - ***Paging Fragments 458***
    - Paging Other Stuff 463
    - Indicators 464
    - Revisiting the Containers Sampler 467
- Tutorial #10 - Rigging Up a ViewPager
    - Step #1: Add a ViewPager to the Layout 475
    - Step #2: Creating a ContentsAdapter 476
    - Step #3: Setting Up the ViewPager 477
    - In Our Next Episode... 480
- Resource Sets and Configurations
    - What's a Configuration? And How Do They Change? 481
    - Configurations and Resource Sets 482
    - Screen Size and Orientation 483
    - Coping with Complexity 486
    - Choosing The Right Resource 487
    - API-Versioned Resources 491
    - ***Default Change Behavior 493***
    - State Saving Scenarios 495
    - Your Options for Configuration Changes 496
    - Blocking Rotations 510
    - And Now, a Word From the Android Project View 511
    - Configuration Challenges 512
- Material Design Basics
    - Your App, in Technicolor! 517
- Dealing with Threads
    - The Main Application Thread 523
    - Getting to the Background 525
    - ***Asyncing Feeling 525***
    - Alternatives to AsyncTask 534
    - And Now, The Caveats 536
    - Event Buses 537
    - Visit the Trails! 545
- Requesting Permissions
    - Frequently-Asked Questions About Permissions 548
    - Characteristics of Permissions 553
    - New Permissions in Old Applications 556

- Android 6.0+ Runtime Permission System 557
- A Simple Runtime Permission Abstraction 569

- Assets, Files, and Data Parsing
  - Packaging Files with Your App 573
  - Files and Android 575
  - Working with Internal Storage 576
  - Working with External Storage 585
  - Removable Storage 592
  - Multiple User Accounts 593
  - Linux Filesystems: You Sync, You Win 593
  - StrictMode: Avoiding Janky Code 595
  - Files, and Your Development Machine 597
  - XML Parsing Options 605
  - JSON Parsing Options 606
  - Using Files with Implicit Intents 606
  - Visit the Trails! 607
- Tutorial #11 - Adding Simple Content
  - Step #1: Adding Some Content 609
  - Step #2: Using SimpleContentFragment 610
  - Step #3: Launching Our Activities, For Real This Time 611
  - Step #4: Getting a Bit More Material 612
  - Step #5: Seeing the Results 614
  - In Our Next Episode... 616
- Tutorial #12 - Displaying the Book
  - Step #1: Adding a Book 617
  - Step #2: Creating a ModelFragment 618
  - Step #3: Defining Our Model 619
  - Step #4: Examining Our Model 621
  - Step #5: Defining Our Event 621
  - Step #6: Loading Our Model 622
  - Step #7: Registering for Events 625
  - Step #8: Adapting the Content 626
  - Step #9: Showing the Content When Loaded 627
  - Step #10: Attaching our ModelFragment 628
  - Step #11: Showing the Content After a Configuration Change 629
  - Step #12: Setting Up StrictMode 630
  - In Our Next Episode... 631
- Using Preferences
  - Getting What You Want 633
  - Stating Your Preference 634
  - Collecting Preferences with PreferenceFragment 635

- Android Studio's Preferences Editor 642
- Types of Preferences 643

- Tutorial #13 - Using Some Preferences
  - Step #1: Defining the Preference XML Files 649
  - Step #2: Creating Our Preference Activity 653
  - Step #3: Adding To Our Action Bar 654
  - Step #4: Launching the Preference Activity 654
  - Step #5: Loading the Preferences 657
  - Step #6: Saving the Last-Read Position 661
  - Step #7: Restoring the Last-Read Position 662
  - Step #8: Keeping the Screen On 663
  - In Our Next Episode... 665
- SQLite Databases
  - Introducing SQLite 667
  - Thinking About Schemas 668
  - Start with a Helper 668
  - Getting Data Out 673
  - The Rest of the CRUD 680
  - Hey, What About Hibernate? 686
  - But, What About Room? 686
  - Visit the Trails! 686
- Tutorial #14 - Saving Notes
  - Step #1: Adding a DatabaseHelper 689
  - Step #2: Examining DatabaseHelper 690
  - Step #3: Creating a NoteFragment 691
  - Step #4: Examining NoteFragment 692
  - Step #5: Creating the NoteActivity 692
  - Step #6: Examining NoteActivity 693
  - Step #7: Add Notes to the Action Bar 694
  - Step #8: Defining a NoteLoadedEvent 696
  - Step #9: Loading a Note from the Database 697
  - Step #10: Loading the Note Into the Fragment 698
  - Step #11: Updating the Database 699
  - Step #12: Saving the Note 700
  - Step #13: Adding a Delete Action Bar Item 702
  - Step #14: Closing the NoteFragment When Deleted 705
  - In Our Next Episode... 710
- Internet Access
  - DIY HTTP 711
  - What About HttpClient? 721
  - HTTP via DownloadManager 724

- Using Third-Party JARs 724
- SSL 725
- Using HTTP Client Libraries 725
- Visit the Trails 751

- Intents, Intent Filters
  - What's Your Intent? 753
  - Stating Your Intent(ions) 755
  - Responding to Implicit Intents 755
  - Requesting Implicit Intents 758
  - ShareActionProvider 762
  - Practice Safe Content Resolution 765
- Broadcasts and Broadcast Receivers
  - The Stopped State 769
  - Example System Broadcasts 770
  - The Order of Things 779
  - Keeping It Local 780
  - Visit the Trails! 780
- Tutorial #15 - Sharing Your Notes
  - Step #1: Adding a ShareActionProvider 781
  - Step #2: Sharing the Note 782
  - Step #3: Testing the Result 784
  - In Our Next Episode... 787
- Services and the Command Pattern
  - Why Services? 789
  - ***Setting Up a Service 790***
  - ***Communicating To Services 793***
  - ***Scenario: The Music Player 795***
  - Communicating From Services 798
  - ***Scenario: The Downloader 801***
  - ***JobIntentService 808***
  - ***IntentService or JobIntentService? 812***
  - Services and Configuration Changes 812
  - ***When Do Services End? 813***
  - ***Background Service Limitations 814***
- Tutorial #16 - Updating the Book
  - Step #1: Adding a Stub DownloadCheckService 819
  - Step #2: Tying the Service Into the Action Bar 820
  - Step #3: Defining Our Event 822
  - Step #4: Defining Our JSON 822
  - Step #5: Defining Our Retrofit Interface 823
  - Step #6: Retrieving Our JSON Via Retrofit 824

    - Step #7: Downloading the Update 826
    - Step #8: Unpacking the Update 827
    - Step #9: Using the Update 831
    - In Our Next Episode... 836
- Large-Screen Strategies and Tactics
    - Objective: Maximum Gain, Minimum Pain 837
    - The Fragment Strategy 837
    - Fragment Example: The List-and-Detail Pattern 846
    - Other Master-Detail Strategies 860
    - Showing More Pages 873
    - Fragment FAQs 879
    - Screen Size and Density Tactics 880
    - Other Considerations 883
- Tutorial #17 - Supporting Large Screens
    - Step #1: Creating Our Layouts 887
    - Step #2: Loading Our Sidebar Widgets 891
    - Step #3: Opening the Sidebar 892
    - Step #4: Loading Content Into the Sidebar 893
    - Step #5: Removing Content From the Sidebar 896
- Backwards Compatibility Strategies and Tactics
    - Think Forwards, Not Backwards 903
    - Aim Where You Are Going 905
    - A Target-Rich Environment 905
    - Lint: It's Not Just For Belly Buttons 906
    - A Little Help From Your Friends 907
    - Avoid the New on the Old 907
    - Testing 911
    - Keeping Track of Changes 911
- System Services
    - What is a System Service? 913
    - What System Services Are There? 914
- Google Play Services
    - What Is Google Play Services? 917
    - What Is In the Play Services SDK? 918
    - Adding Play Services to Your Project 925
- Getting Help
    - Questions. Sometimes, With Answers. 935
    - Heading to the Source 936
- Working with Library Modules
    - Prerequisites 939
    - Creating a Library Module 939

    - Using a Library Module 942
    - Library Modules and the Manifest 942
    - Library Modules and Transitive Dependencies 943
    - Limitations of Library Modules 944
- Gradle and Tasks
    - Key Build-Related Tasks 945
    - Results 948
- Gradle Build Variants
    - Prerequisites 949
    - Objectives of the Project Structure 949
    - Terminology 950
    - Configuring the Stock Build Types 953
    - Adding Build Types 957
    - Adding Product Flavors and Getting Build Variants 957
    - Doing the Splits 959
    - Gradle and Android Studio 961
- Manifest Merger Rules
    - Prerequisites 963
    - Manifest Scenarios 963
    - Pieces of Manifest Generation 965
    - Examining the Merger Results 967
    - Viewing Merged Manifests in Android Studio 967
    - Merging Elements and Attributes 968
    - Employing Placeholders 978
- Signing Your App
    - Prerequisites 983
    - Role of Code Signing 983
    - What Happens In Debug Mode 984
    - Production Signing Keys 985
- Distribution
    - Prerequisites 993
    - Get Ready To Go To Market 993
- Writing a Gradle Plugin
    - Prerequisites 999
    - Customizing a Gradle Build 999
    - Some Use Cases for a Custom Plugin 1000
    - Writing a Plugin 1001
    - Distributing the Plugin 1008
    - Using the Plugin 1010
    - Creating a Real Plugin 1014
- Code Generation

    - Prerequisites 1019
    - What Drives the Custom Code? 1019
    - Java as Poetry 1021
    - Writing a Code Generation Plugin 1022
    - Using the Generated Code 1033
- Advanced Gradle for Android Tips
    - Prerequisites 1035
    - Gradle, DRY 1035
    - Automating APK Version Information 1038
    - ***Adding to BuildConfig 1040***
- Testing with JUnit4
    - Prerequisites 1045
    - Instrumentation Tests and Unit Tests 1045
    - Writing JUnit4 Test Cases 1047
    - Configuring Gradle 1053
    - Running Your Instrumentation Tests 1054
    - Testing Android Library Projects 1059
    - Testing and Runtime Permissions 1059
    - The Android Test Orchestrator 1060
- Testing with Espresso
    - Prerequisites 1065
    - Adding a Shot of Espresso 1066
    - Writing Tests in Espresso 1068
    - The Espresso Test Recorder 1074
    - Stronger Espresso 1079
    - Opting Out of Analytics 1093
    - Waiting for the World to Change 1094
- Testing with UI Automator
    - Prerequisites 1099
    - What Is UI Automator? 1099
    - Why Choose UI Automator Over Alternatives? 1100
    - Gradle and Android Studio Settings 1100
    - Creating a Test Case 1101
    - Running Your Tests 1109
    - Finding Your Widgets 1109
    - Using the UI Automator Viewer 1110
- Measuring Test Coverage
    - Prerequisites 1113
    - Who Tests the Testers? 1113
    - Some Types of Test Coverage 1114
    - Coverage and Your Instrumentation Tests 1116

- Unit Testing
  - Prerequisites 1121
  - I Thought We Were Already Unit Testing? 1122
  - Scenario: Clean Architecture 1122
  - Setting Up Unit Testing 1123
  - Writing POJO Unit Tests 1126
  - Running Unit Tests 1130
  - ***Mocking Android 1135***
- MonkeyRunner and the Test Monkey
  - Prerequisites 1151
  - MonkeyRunner 1151
  - Monkeying Around 1153
- Java 8 Lambda Expressions
  - Prerequisites 1159
  - The Basic Idea 1160
  - Using Lambda Expressions 1161
  - Alternative: Method References 1165
- Rx Basics
  - Prerequisites 1167
  - Life is But a Stream 1167
  - Action and Reaction 1169
  - A Rx For What Ails You 1170
  - Rx and Lambdas 1171
  - ***A Simple Stream 1171***
  - Be Your Own Stream 1173
  - ***Removing the AsyncTask 1175***
  - Lambdas and Lifetimes 1181
  - Streaming from a Resource 1182
  - Error Handling 1183
  - Transmogrification 1184
  - ***Rx-Enabled Libraries 1185***
  - Further Reading 1187
  - What About LiveData? 1188
- Notifications
  - Prerequisites 1191
  - What's a Notification? 1191
  - Notifications and Channels 1194
  - Showing a Simple Notification 1194
  - The Activity-Or-Notification Scenario 1200
  - Big (and Rich) Notifications 1201
  - Foreground Services 1208

- Disabled Notifications 1211
- Advanced Notifications
  - Prerequisites 1215
  - Being a Good Citizen 1215
  - More About Channels 1216
  - Wear? There! 1223
  - Stacking Notifications 1228
  - Avoiding Wear 1234
  - Other Wear-Specific Notification Options 1235
  - Remote Input, On-Device 1249
  - Notification Groups 1254
  - Lockscreen Notifications 1256
  - Priority, Importance, and Heads-Up Notifications 1265
  - Full-Screen Notifications 1268
  - Progress Notifications 1270
  - Custom Views 1279
  - Life After Delete 1282
  - The Mysterious Case of the Missing Number 1283
  - Notifications and MessagingStyle 1283
  - Changes in API Level 23 1288
  - Sounds and Android 7.0 1289
  - Auto-Timeout 1290
  - Colorized Notifications 1291
- Multi-Window Support
  - Prerequisites 1295
  - A History of Windows 1295
  - What The User Sees 1296
  - What Your Code Sees 1298
  - Opting Out 1300
  - Opting In 1300
  - Configuring the Layout 1301
  - Avoiding Stutter 1303
  - Managing the Background 1304
  - How Low Can You Go? 1304
  - Parallel Processing 1305
  - Split-Screen, HOME, and Your Activity 1307
  - Split-Screen and Orientations 1308
  - Forcing Your App Into Multi-Window/Multi-Instance 1308
  - Supporting Legacy Proprietary Multi-Window 1316
  - Freeform Multi-Window Mode 1319
  - Picture-in-Picture 1326

- Multi-Display Support 1330
- Advanced ConstraintLayout
  - Prerequisites 1339
  - Disclosing Your Bias 1339
  - Centering Yourself 1345
  - Keeping Things Proportional 1349
  - Constraining the ConstraintLayout Size 1350
  - Chains, Without the Whips 1351
  - What If We're GONE? 1355
- GridLayout
  - Prerequisites 1357
  - Issues with the Classic Containers 1357
  - The New Contender: GridLayout 1359
  - GridLayout and the Android Support Package 1359
  - Our Test App 1360
  - Replacing the Classics 1363
  - Implicit Rows and Columns 1369
  - Row and Column Spans 1370
- Dialogs and DialogFragments
  - Prerequisites 1377
  - DatePickerDialog and TimePickerDialog 1377
  - AlertDialog 1383
  - DialogFragments 1384
  - DialogFragment: The Other Flavor 1389
  - Dialogs: Modal, Not Blocking 1389
- Advanced ListViews
  - Prerequisites 1391
  - Multiple Row Types, and Self Inflation 1391
  - Choice Modes and the Activated Style 1398
  - Custom Mutable Row Contents 1399
  - From Head To Toe 1404
  - Enter RecyclerView 1408
- Action Modes
  - Prerequisites 1410
  - A Matter of Context 1410
  - Manual Action Modes 1411
  - Multiple-Choice-Modal Action Modes 1416
  - Long-Click To Initiate an Action Mode 1420
- Other Advanced Action Bar Techniques
  - Prerequisites 1427
  - Action Layouts 1427

- Action Views and Action Providers 1428
- Searching with SearchView 1429
- Floating Action Bars 1435

- Toolbar
  - Prerequisites 1439
  - Basic Toolbar Mechanics 1439
  - Use Case #1: Split Action Bar 1440
  - Use Case #2: Contextual Actions 1446
  - Use Case #3: Replacement Action Bar 1456
- AppCompat: The Official Action Bar Backport
  - Prerequisites 1459
  - Ummmm... Why? 1459
  - The Basics of Using AppCompat 1462
  - Other AppCompat Effects 1469
  - Toolbar and AppCompat 1475
  - To Material, or Not to Material 1479
- The Android Design Support Library
  - Prerequisites 1482
  - GUIs and the Support Package 1482
  - Adding the Library... and What Comes With It 1483
  - Introducing CWAC-CrossPort 1483
  - Snackbars: Sweeter than Toasts 1484
  - Absolutely FABulous 1492
  - Material Tabs with TabLayout 1510
  - Floating Labels 1522
- Implementing a Navigation Drawer
  - Prerequisites 1533
  - What is a Navigation Drawer? 1533
  - A Simple Navigation Drawer 1535
  - Alternative Row Layouts 1541
  - Additional Considerations 1543
  - What Should Not Be in the Drawer 1552
  - Independent Implementations 1553
- RecyclerView
  - Prerequisites 1556
  - AdapterView and its Discontents 1556
  - Enter RecyclerView 1556
  - A Trivial List 1557
  - ***Divider Options 1565***
  - Handling Click Events 1573
  - What About Cursors? 1580

    - Grids 1587
    - Varying the Items 1591
    - Mutable Row Contents 1604
    - Changing the Contents 1632
    - The Order of Things 1638
    - Other Bits of Goodness 1646
    - Animating the Deltas Using DiffUtil 1647
    - The March of the Libraries 1655
- Advanced RecyclerView
    - Prerequisites 1663
    - RecyclerView as Pager 1663
    - Declaring a LayoutManager in the Layout 1686
    - Transcript Mode 1686
- Advanced Uses of WebView
    - Prerequisites 1691
    - Friends with Benefits 1691
    - Navigating the Waters 1706
    - Settings, Preferences, and Options (Oh, My!) 1706
    - Security and Your WebView 1707
    - Android 8.0 WebView Changes 1710
    - Chrome Custom Tabs 1713
    - Prerequisites 1715
    - Keyboards, Hard and Soft 1715
    - Tailored To Your Needs 1716
    - Tell Android Where It Can Go 1721
    - Fitting In 1723
    - Jane, Stop This Crazy Thing! 1726
- Fonts and Text
    - Prerequisites 1727
    - Love The One You're With 1727
    - Yeah, But Do We Really Have To Do This in Java? 1731
    - Here a Glyph, There a Glyph 1732
    - Auto-Sizing TextView 1733
    - Justified Text 1738
- Rich Text
    - Prerequisites 1739
    - The Span Concept 1739
    - ***Loading Rich Text 1741***
    - Editing Rich Text 1744
    - Saving Rich Text 1748
    - Manipulating Rich Text 1748

- Animators
  - Prerequisites 1751
  - ViewPropertyAnimator 1751
  - The Foundation: Value and Object Animators 1756
  - Animating Custom Types 1759
  - Hardware Acceleration 1760
  - The Three-Fragment Problem 1761
- Legacy Animations
  - Prerequisites 1773
  - It's Not Just For Toons Anymore 1773
  - A Quirky Translation 1774
  - Fading To Black. Or Some Other Color. 1778
  - When It's All Said And Done 1780
  - Loose Fill 1781
  - Hit The Accelerator 1782
  - Animate. Set. Match. 1782
  - Active Animations 1784
- Custom Drawables
  - Prerequisites 1785
  - Where Do These Things Go? 1786
  - ColorDrawable 1787
  - AnimationDrawable 1787
  - StateListDrawable 1791
  - ColorStateList 1792
  - LayerDrawable 1794
  - TransitionDrawable 1795
  - LevelListDrawable 1796
  - ScaleDrawable and ClipDrawable 1798
  - InsetDrawable 1807
  - ShapeDrawable 1808
  - BitmapDrawable 1819
  - Composite Drawables 1826
  - A Stitch In Time Saves Nine 1830
- Mapping with Maps V2
  - Prerequisites 1841
  - A Brief History of Mapping on Android 1842
  - Where You Can Use Maps V2 1842
  - Licensing Terms for Maps V2 1843
  - What You Need to Start 1843
  - The Book Samples... And You! 1846
  - Setting Up a Basic Map 1846

    - Playing with the Map 1851
    - Map Tiles 1854
    - Placing Simple Markers 1854
    - Seeing All the Markers 1858
    - Flattening and Rotating Markers 1860
    - Sprucing Up Your “Info Windows” 1864
    - Images and Your Info Window 1869
    - Setting the Marker Icon 1876
    - Responding to Taps 1878
    - Dragging Markers 1879
    - The “Final” Limitations 1882
    - A Bit More About IPC 1885
    - Finding the User 1885
    - Dealing with Runtime Permissions 1890
    - Drawing Lines and Areas 1893
    - Gestures and Controls 1896
    - Tracking Camera Changes 1897
    - Maps in Fragments and Pagers 1900
    - Animating Marker Movement 1906
    - Maps, of the Indoor Variety 1912
    - Taking a Snapshot of a Map 1913
    - MapFragment vs. MapView 1914
    - About That AbstractMapActivity Class... 1914
    - Helper Libraries for Maps V2 1918
    - Problems with Maps V2 at Runtime 1922
    - Problems with Maps V2 Deployment 1922
    - What Non-Compliant Devices Show 1922
    - Mapping Alternatives 1923
- Crafting Your Own Views
    - Prerequisites 1925
    - Pick Your Poison 1925
    - Colors, Mixed How You Like Them 1926
    - ReverseChronometer: Simply a Custom Subclass 1937
    - AspectLockedFrameLayout: A Custom Container 1943
    - Mirror and MirroringFrameLayout: Draw It Yourself 1946
- Advanced Preferences
    - Prerequisites 1959
    - Introducing PreferenceActivity 1959
    - Intents for Headers or Preferences 1964
    - Conditional Headers 1965
    - Dependent Preferences 1970

- Nested Screens 1973
- Listening to Preference Changes 1976
- Defaults, and Defaults 1979
- Listening to Preference Value Changes 1980
- ***Dynamic ListPreference Contents 1980***
- ***Dealing with External Changes to Preferences 1985***
- Preferences in Device Settings App 1986

- Custom Dialogs and Preferences
  - Prerequisites 1995
  - Your Dialog, Chocolate-Covered 1995
  - Preferring Your Own Preferences, Preferably 1999
- Progress Indicators
  - Prerequisites 2007
  - Progress Bars 2007
  - ProgressBar and Threads 2011
  - Tailoring Progress Bars 2014
  - Progress Dialogs 2024
  - Title Bar and Action Bar Progress Indicators 2026
  - Direct Progress Indication 2028
- More Fun with Pagers
  - Prerequisites 2031
  - ***Hosting ViewPager in a Fragment 2031***
  - Pages and the Action Bar 2033
  - ViewPagers and Scrollable Contents 2036
  - ***Columns for Large, Pages for Small 2036***
- Focus Management and Accessibility
  - Prerequisites 2043
  - Prepping for Testing 2044
  - Controlling the Focus 2044
  - Accessibility and Focus 2054
  - Accessibility Beyond Focus 2055
  - Accessibility Beyond Impairment 2065
- Miscellaneous UI Tricks
  - Prerequisites 2067
  - Full-Screen and Lights-Out Modes 2067
  - Offering a Delayed Timeout 2078
- Event Bus Alternatives
  - Prerequisites 2083
  - A Brief Note About the Sample Apps 2083
  - Standard Intents as Event Bus 2083
  - ***LocalBroadcastManager as Event Bus 2084***

    - ***greenrobot's EventBus 3.x 2095***
    - Hey, What About Otto? 2100
- Tasks
    - Prerequisites 2103
    - First, Some Terminology 2103
    - And Now, a Bit About Task Killers 2109
    - A Canary for the Task's Coal Mine 2116
    - The Default User Experience 2119
    - Explaining the Default Behavior 2123
    - Basic Scenarios for Changing the Behavior 2126
    - Dealing with the Persistent Tasks 2141
    - Documents As Tasks 2143
    - Other Task-Related Activity Properties 2146
    - Other Task-Related Activity Methods 2155
- The Assist API ("Now On Tap")
    - Prerequisites 2158
    - What Data Gets Disclosed 2158
    - Adding to the Data 2160
    - Removing from the Data 2163
    - Blocking Assist as a User 2166
    - Implementing Your Own Assistant 2169
- The Autofill API
    - Prerequisites 2184
    - The Pieces of the Puzzle 2184
    - The User Experience 2185
    - What Data Gets Disclosed 2187
    - Blocking Autofill as a User 2201
    - Supporting Autofill with Standard Widgets 2203
    - Supporting Autofill with Custom Widgets 2206
    - Dealing with Dynamic Changes 2206
    - Security Requirements of Autofill Services 2207
- The Data Binding Framework
    - Prerequisites 2213
    - The What, Now? 2214
    - The Basic Steps 2215
    - The Extended Layout Resource 2221
    - The Binding Expression Language 2225
    - Observables and Updating the Binding 2228
    - Two-Way Binding 2241
    - Other Features of Note 2244
- Drag and Drop

    - Prerequisites 2273
    - The Scope of Drag and Drop 2273
    - The Pieces of Drag-and-Drop 2275
    - Drag-and-Drop, within an Activity 2279
    - Drag-and-Drop, Between Apps 2288
    - Detecting Cross-App Drag Events 2300
    - Intra-App Cross-Window Drag-and-Drop 2300
    - Pondering Legacy Multi-Window 2303
    - Dragging and Dropping Simple Stuff 2303
    - Multi-Action Drag-and-Drop 2304
    - Nested Drop Targets 2310
    - Pondering Standards 2319
    - Pondering Accessibility 2319
- Keyboard and Mouse Input
    - Prerequisites 2321
    - Offering Keyboard Shortcuts 2321
    - Custom Copy-and-Paste 2331
    - Physical Keyboards and Focusing 2335
    - Offering Mouse Context Menus 2339
    - Offering Tooltips 2345
    - Pointer Capture 2353
- Viewing PDFs
    - Prerequisites 2355
    - The Criteria 2356
    - The Classic Solution: ACTION_VIEW 2358
    - The Really Bad Idea: Google Docs 2358
    - The Built-In Option: PdfRenderer 2359
    - The Thunder Lizard Choice: PDF.js 2368
    - The Native Approach: Pdfium 2372
    - What To Choose? 2376
- Home Screen App Widgets
    - Prerequisites 2379
    - App Widgets and Security 2379
    - The Big Picture for a Small App Widget 2380
    - ***Crafting App Widgets 2381***
    - Another and Another 2388
    - App Widgets: Their Life and Times 2389
    - Controlling Your (App Widget's) Destiny 2389
    - ***One Size May Not Fit All 2390***
    - Lockscreen Widgets 2396
    - Preview Images 2402

- Being a Good Host 2404
- Adapter-Based App Widgets
  - Prerequisites 2405
  - AdapterViews for App Widgets 2405
  - ***Building Adapter-Based App Widgets 2406***
- Advanced Permissions
  - ***Prerequisites 2421***
  - Securing Yourself 2421
  - Signature Permissions 2424
  - The Custom Permission Vulnerability 2426
  - Custom Dangerous Permissions, and Android 6.0 2433
  - Finding the Available Permissions 2437
- Restricted Profiles and UserManager
  - Prerequisites 2451
  - Android Tablets and Multiple User Accounts 2451
  - Determining What the User Can Do 2457
  - Impacts of Device-Level Restrictions 2460
  - Enabling Custom Restrictions 2460
  - Implicit Intents May Go “Boom” 2471
- Device Authentication
  - ***Prerequisites 2473***
  - ***Is the Device Secure? 2473***
  - ***Reconfirming the User 2478***
  - ***Fingerprints 2480***
- Keys and the Keystore
  - ***Prerequisites 2496***
  - ***Terminology 2496***
  - ***Getting a KeyStore 2497***
  - ***Creating a Key 2498***
  - ***Tying the Key to Device Authentication 2499***
  - ***Learning More About Your Key 2500***
  - ***Encrypting Data 2500***
  - ***Time-Limited Device Authentication 2511***
  - ***Encrypting Passphrases 2514***
  - ***A Key(Store) Limitation 2523***
- Miscellaneous Security Techniques
  - Prerequisites 2525
  - Public Key Validation 2525
  - Choosing Your Signing Keysize 2537
  - Avoiding Accidental APIs 2538
  - Other Ways to Expose Data 2543

    - Jacking Attacks 2544
    - Using FLAG_SECURE 2552
- Content Provider Theory
    - Prerequisites 2557
    - Using a Content Provider 2557
    - Building Content Providers 2565
    - Issues with Content Providers 2573
- Content Provider Implementation Patterns
    - Prerequisites 2575
    - The Single-Table Database-Backed Content Provider 2575
    - The Local-File Content Provider 2583
    - The Protected Provider 2592
    - The Stream Provider 2594
    - FileProvider 2598
    - StreamProvider 2603
- The Loader Framework
    - Prerequisites 2613
    - Cursors: Issues with Management 2614
    - Introducing the Loader Framework 2614
    - Choosing an Implementation 2616
    - Using CursorLoader 2616
    - What Else Is Missing? 2619
    - What Happens When...? 2619
    - Writing a Custom Loader 2620
- The ContactsContract and CallLog Providers
    - Prerequisites 2629
    - Introducing You to Your Contacts 2630
    - Pick a Peck of Pickled People 2631
    - Spin Through Your Contacts 2633
    - Makin' Contacts 2640
    - Looking at the CallLog 2644
- The CalendarContract Provider
    - Prerequisites 2650
    - You Can't Be a Faker 2650
    - Do You Have Room on Your Calendar? 2650
    - Penciling In an Event 2656
- The MediaStore Provider
    - Prerequisites 2659
    - What Is the MediaStore? 2660
    - MediaStore and "Other" External Storage 2661
    - How Does My Content Get Indexed? 2662

    - How Do I Retrieve Video from the MediaStore? 2662
- Consuming Documents
    - Prerequisites 2671
    - The Storage Access... What? 2671
    - The Storage Access Framework Participants 2673
    - Picking How to Pick (a Peck of Pickled Pepper Photos) 2673
    - Opening a Document 2674
    - Why We Want Things To Be Openable 2678
    - The Rest of the CRUD 2679
    - The DocumentFile Helper 2680
    - CWAC-Document and DocumentFileCompat 2680
    - Getting Durable Access 2681
    - Another Durable Example: Diceware 2691
    - ***Extended Example: A Tiny Text Editor 2705***
    - Document Trees 2735
    - Getting a Tree: Example 2737
    - Scoped Directory Access Bug 2754
    - Android 8.0 Changes 2755
- Providing Documents
    - Prerequisites 2757
    - Have Your Content, and Provide it Too 2757
    - Key Provider Concepts 2759
    - Pieces of a Provider 2760
    - Optional Provider Capabilities 2775
- Encrypted Storage
    - Prerequisites 2782
    - Scenarios for Encryption 2782
    - Obtaining SQLCipher 2782
    - Using SQLCipher 2783
    - SQLCipher Limitations 2784
    - Passwords and Sessions 2785
    - About Those Passphrases... 2786
    - Encrypted Preferences 2793
    - IOCipher 2795
- Packaging and Distributing Data
    - Prerequisites 2797
    - Packing a Database To Go 2797
- Advanced Database Techniques
    - Prerequisites 2801
    - Full-Text Indexing 2801
- Data Backup

    - Prerequisites 2819
    - First, Some Terminology 2820
    - Differing Definitions of "Backup" 2820
    - ***Implementing IT-Style Backup 2822***
    - ***The Google Backup Bootstrap 2847***
    - Boosting Backup Security 2856
    - Alternative Approaches 2857
- SSL
    - Prerequisites 2863
    - Basic SSL Operation 2863
    - Problems in Paradise 2864
    - ***Introducing Network Security Configuration 2867***
    - ***SSL Problems and Network Security Configuration 2870***
    - Other SSL Strengthening Techniques 2881
    - Advanced Uses of CWAC-NetSecurity 2883
    - NetCipher 2886
- NetCipher
    - Prerequisites 2887
    - Network Security's Got Onions 2887
    - The NetCipher HTTP Integration APIs 2889
    - The Rest of the Builder API 2896
- Embedding a Web Server
    - Prerequisites 2909
    - Why a Web Server? 2909
    - Introducing AsyncHttpServer 2911
    - Embedding a Simple Server 2911
    - Template-Driven Responses, with Handlebars 2928
    - Supporting WebSockets 2933
    - Securing the Web Server 2939
    - Towards a Reusable Web Server Service 2946
- Miscellaneous Network Topics
    - Prerequisites 2957
    - Downloading Files 2957
    - Data Saver 2971
- Audio Playback
    - Prerequisites 2975
    - Get Your Media On 2975
    - MediaPlayer for Audio 2976
    - Other Ways to Make Noise 2981
- Audio Recording
    - Prerequisites 2985

    - Recording by Intent 2985
    - Recording to Files 2988
    - Recording to Streams 2993
    - Raw Audio Input 2996
    - Requesting the Microphone 2996
- Video Playback
    - Prerequisites 2997
    - Moving Pictures 2997
- Using the Camera via 3rd-Party Apps
    - Prerequisites 3003
    - Being Specific About Features 3003
    - Still Photos: Letting the Camera App Do It 3004
    - Permissions and Third-Party Camera Apps 3011
    - A Matter of Orientation 3012
    - Scanning with ZXing 3019
    - Videos: Letting the Camera App Do It 3020
    - Using a Camera Library 3023
    - Directly Working with the Camera 3035
- Working Directly with the Camera
    - Prerequisites 3037
    - Notes About the Code Snippets 3037
    - A Tale of Two APIs 3038
    - Performing Basic Camera Operations 3039
    - Configuring the Still Camera 3062
    - And Now, The Problems 3071
- Media Routes
    - Prerequisites 3077
    - Terminology 3077
    - A Tale of Two MediaRouters 3078
    - Attaching to MediaRouter 3079
    - User Route Selection with MediaRouteActionProvider 3081
    - Using Live Video Routes 3096
    - Using Remote Playback Routes 3096
- Supporting External Displays
    - Prerequisites 3115
    - A History of External Displays 3115
    - What is a Presentation? 3116
    - Playing with External Displays 3117
    - Detecting Displays 3123
    - A Simple Presentation 3124
    - A Simpler Presentation 3130

    - Presentations and Configuration Changes 3136
    - Presentations as Fragments 3137
    - Another Sample Project: Slides 3148
    - Device Support for Presentation 3156
    - ***Presentations from a Service 3157***
    - Hey, What About Chromecast? 3160
- Google Cast and Chromecast
    - Prerequisites 3163
    - Here a Cast, There a Cast 3163
    - Common Chromecast Development Notes 3165
    - Your API Choices 3165
    - Senders and Receivers 3166
    - Supported Media Types 3167
    - Cast SDK Dependencies 3168
    - Developing Google Cast Apps 3170
- The "Ten-Foot UI"
    - Prerequisites 3171
    - What is the "Ten-Foot UI"? 3172
    - Overscan 3172
    - Navigation 3173
    - Stylistic Considerations 3174
    - The Leanback UI 3176
    - Testing Your Theories 3191
- Putting the TVs All Together: Decktastic
    - Prerequisites 3194
    - Introducing Decktastic 3194
    - Implementing Decktastic 3198
- Creating a MediaRouteProvider
    - Prerequisites 3233
    - Terminology 3233
    - DIY Chromecast 3234
    - Creating the MediaRouteProvider 3236
    - Consuming the MediaRouteProvider 3247
    - Implementing This "For Realz" 3251
- The Media Projection APIs
    - Prerequisites 3253
    - ***Requesting Screenshots 3253***
    - Recording the Screen 3268
    - Yet Another Sample: andshooter 3288
- AlarmManager and the Scheduled Service Pattern
    - Scenarios 3295

    - Options 3296
    - A Simple Example 3298
    - The Five set...() Varieties 3300
    - The Four Types of Alarms 3302
    - When to Schedule Alarms 3302
    - ***Archetype: Scheduled Service Polling 3304***
    - Warning: Not All Android Devices Play Nice 3310
    - Debugging Alarms 3310
    - Android 6.0 and the War on Background Processing 3313
    - Android 7.0 and OnAlarmListener 3313
- PowerManager and WakeLocks
    - Prerequisites 3317
    - Keeping the Screen On, UI-Style 3317
    - The Role of the WakeLock 3318
    - What WakefulIntentService Does 3319
- JobScheduler
    - Prerequisites 3321
    - The Limitations of AlarmManager 3321
    - Enter the JobScheduler 3322
    - ***Employing JobScheduler 3322***
    - Pondering Backoff Criteria 3336
    - Other JobScheduler Features 3338
    - JobScheduler Period Limits 3339
    - GcmNetworkManager 3339
    - ***Periodic Work, Across Device Versions 3339***
    - ***Android 6.0 and "the War on Background Processing" 3344***
    - ***Scheduling Content Monitoring 3356***
- Accessing Location-Based Services
    - Prerequisites 3371
    - Location Providers: They Know Where You're Hiding 3372
    - Finding Yourself 3372
    - On the Move 3374
    - Getting Locations via PendingIntent 3382
    - Are We There Yet? Are We There Yet? Are We There Yet? 3382
    - Testing... Testing... 3383
    - Alternative Flavors of Updates 3384
    - The Fused Option 3385
    - Locations and Features 3385
- The Fused Location Provider
    - Prerequisites 3387
    - Why Use the Fused Location Provider? 3387

    - Why Not Use the Fused Location Provider? 3388
    - Finding Our Location, Once 3388
    - Requesting Location Updates 3393
    - I Can Haz Location? 3395
- Working with the Clipboard
    - Prerequisites 3405
    - Working with the Clipboard 3405
    - ClipData and Drag-and-Drop 3410
    - Monitoring the Clipboard 3411
    - The Android 4.3 Clipboard Bug 3412
- Telephony
    - Prerequisites 3415
    - Report To The Manager 3416
    - You Make the Call! 3416
    - No, Really, You Make the Call! 3419
- Working With SMS
    - Prerequisites 3421
    - Sending Out an SOS, Give or Take a Letter 3422
    - Monitoring and Receiving SMS 3430
    - The SMS Inbox 3436
    - Asking to Change the Default 3437
    - SMS and the Emulator 3438
    - SMS Tokens 3439
- NFC
    - Prerequisites 3445
    - What Is NFC? 3445
    - To NDEF, Or Not to NDEF 3447
    - NDEF Modalities 3447
    - NDEF Structure and Android's Translation 3448
    - The Reality of NDEF 3449
    - Sources of Tags 3451
    - Writing to a Tag 3451
    - Responding to a Tag 3460
    - Expected Pattern: Bootstrap 3461
    - Mobile Devices are Mobile 3462
    - Enabled and Disabled 3462
    - Android Beam 3462
    - Beaming Files 3470
    - Another Sample: SecretAgentMan 3471
    - Additional Resources 3481
- Device Administration

    - Prerequisites 3483
    - Objectives and Scope 3483
    - Defining and Registering an Admin Component 3484
    - Going Into Lockdown 3490
    - Passwords and Device Administration 3497
    - Getting Along with Others 3502
- Basic Use of Sensors
    - Prerequisites 3503
    - The Sensor Abstraction Model 3503
    - Considering Rates 3504
    - Reading Sensors 3505
    - Batching Sensor Readings 3516
- Printing and Document Generation
    - Prerequisites 3518
    - The Android Print System 3518
    - About the Sample App 3519
    - Printing a Bitmap 3520
    - Printing an HTML Document 3522
    - Printing a PDF File 3527
    - Printing Using a Canvas 3535
    - Print Jobs 3538
    - Printing, Threads, and Services 3538
    - Printing Prior to Android 4.4 3541
    - HTML Generation 3541
    - PDF Generation Options 3545
- Basic Bluetooth RFCOMM
    - Prerequisites 3547
    - A Quick Bit of Scope 3547
    - About the Sample App 3548
    - Bluetooth and Permissions 3554
    - The Rx for Your Bluetooth 3555
    - I Can Haz Bluetooth? 3556
    - I Feel a Bond Between Us 3559
    - A Voyage of Discovery 3562
    - Serving and Shouting 3565
    - Reach Out and Touch Someone 3570
    - Ping and Pong 3574
    - Differences with Android Things 3578
- Dealing with Different Hardware
    - Prerequisites 3581
    - Filtering Out Devices 3581

    - Runtime Capability Detection 3584
    - Dealing with Device Bugs 3585
- Writing and Using Parcelables
    - Prerequisites 3589
    - The Role of Parcelable 3589
    - Writing a Parcelable 3590
    - ***The Limitations of Parcelable 3596***
    - ***Beware the PendingIntent 3599***
- Responding to URLs
    - Prerequisites 3605
    - Manifest Modifications 3605
    - Creating a Custom URL 3607
    - Reacting to the Link 3608
    - App Links 3610
- App Shortcuts
    - Prerequisites 3621
    - Enabling Deep Dives 3622
    - App Shortcuts, from the User's POV 3622
    - Offering Manifest App Shortcuts 3624
    - Offering Dynamic App Shortcuts 3630
    - Privacy, Security, and App Shortcuts 3642
- PackageManager Tricks
    - Prerequisites 3643
    - Asking Around 3643
    - Preferred Activities 3647
    - Middle Management 3653
- Remote Services and the Binding Pattern
    - Prerequisites 3657
    - The Binding Pattern 3658
    - When IPC Attacks! 3664
    - Service From Afar 3667
    - Tightening Up the Security 3673
    - Servicing the Service 3677
    - Thinking About Security 3682
    - The "Everlasting Service" Anti-Pattern 3682
- Advanced Manifest Tips
    - Prerequisites 3685
    - Just Looking For Some Elbow Room 3685
    - Using an Alias 3695
    - Getting Meta (Data) 3696
- Miscellaneous Integration Tips

- Prerequisites 3701
- ***Direct Share 3701***
- Take the Shortcut 3710
- Homing Beacons for Intents 3717
- Integrating with Text Selection 3717
- ***Quick Settings and TileService 3728***
- Installing Packages 3737
- Deleting Packages 3740
- Detecting Changes in Packages 3740

• Replacing App Code Dynamically
- Prerequisites 3743
- Typical Objectives 3744
- The Challenges 3746
- ***The Scripting Solution 3748***
- The Hybrid Solution 3749
- The Patch Solution 3750
- A DIY Solution 3751
- Is Any Of This a Good Idea? 3774

• Android Studio Editors and Dialogs
- Prerequisites 3775
- Project Structure 3775
- Translations Editor 3786

• Advanced Emulator Capabilities
- Prerequisites 3793
- Other Notable Configuration Options 3793
- The Emulator Sidebar 3799
- Emulator Window Operations 3810
- Headless Operation 3810

• Lint and the Support Annotations
- Prerequisites 3813
- What It Is 3814
- When It Runs 3814
- What to Fix 3815
- What to Configure 3816
- Support Annotations 3820

• Inspecting Layouts
- Prerequisites 3833
- The Layout Inspector 3833
- Hierarchy View 3835

• Screenshots and Screencasts
- Prerequisites 3841

    - Collecting from Android Studio 3841
    - Screencasts 3845
    - Collecting from the Command Line 3845
    - Collecting from Another App 3847
    - Tips and Tricks 3848
- ADB Tips and Tricks
    - Prerequisites 3849
    - This is the Droid That You Are Looking For 3849
    - Installing and Uninstalling Apps 3850
    - Playing with Permissions 3850
    - Starting and Stopping Components 3851
    - Killing Processes and Clearing Data 3852
    - Changing Display Metrics 3852
- Stetho
    - Wait, Wut? Chrome? 3855
    - Basic Stetho Integration 3855
    - Connecting Chrome to Your App 3862
    - What You Get In Chrome Dev Tools 3863
    - Getting Help with Stetho 3871
- Issues with Speed
    - Prerequisites 3875
    - Getting Things Done 3875
    - Your UI Seems... Janky 3876
    - Not Far Enough in the Background 3876
    - Playing with Speed 3877
- Finding CPU Bottlenecks
    - Prerequisites 3879
    - Android Studio's Profiler 3880
    - Method Tracing 3882
    - Other General CPU Measurement Techniques 3887
    - UI "Jank" Measurement 3888
- Focus On: NDK
    - Prerequisites 3905
    - The Role of the NDK 3906
    - Introducing CWAC-AndDown 3909
    - Installing the NDK 3909
    - The Contents of an NDK Project 3910
    - Building Your Library 3913
    - libhoudini and the NDK 3919
- Improving CPU Performance in Java
    - Prerequisites 3921

    - Reduce CPU Utilization 3921
    - Reduce Time on the Main Application Thread 3926
    - Improve Throughput and Responsiveness 3934
- Finding and Eliminating Jank
    - Prerequisites 3937
    - The Case: ThreePaneDemoBC 3937
    - Are We Janky? 3938
    - Finding the Source of the Jank 3938
    - Where Things Went Wrong 3949
    - Removing the Jank 3949
    - Frame Metrics API 3950
- Issues with Bandwidth
    - Prerequisites 3957
    - You're Using Too Much of the Slow Stuff 3958
    - You're Using Too Much of the Expensive Stuff 3958
    - You're Using Too Much of Somebody Else's Stuff 3959
    - You're Using Too Much... And There Is None 3960
- Focus On: TrafficStats
    - Prerequisites 3961
    - TrafficStats Basics 3961
    - Example: TrafficMonitor 3963
    - Other Ways to Employ TrafficStats 3972
- Measuring Bandwidth Consumption
    - Prerequisites 3975
    - On-Device Measurement 3975
    - Off-Device Measurement 3979
    - Android Studio Profiler 3981
- Being Smarter About Bandwidth
    - Prerequisites 3983
    - Bandwidth Savings 3983
    - Bandwidth Shaping 3987
    - Avoiding Metered Connections 3991
    - Data Saver 3991
- Issues with Application Heap
    - Prerequisites 3993
    - You Are in a Heap of Trouble 3994
    - Determining Your Heap Size At Runtime 3995
    - Fragments of Memory 3995
    - Getting a Trim 3996
    - Warning: Contains Graphic Images 3997
    - Releasing SQLite Memory 4009

    - Cheating 4009
    - The 1MB IPC Transaction Limit 4010
- Finding Memory Leaks
    - Prerequisites 4013
    - Android Studio Profiler 4014
    - Getting Heap Dumps 4017
    - Analyzing Heap Dumps in Android Studio 4018
    - Common Leak Scenarios 4024
    - A Canary in a Leaky Coal Mine 4031
- Issues with System RAM
    - Prerequisites 4041
    - Can't We All Just Get Along? 4041
    - Contributors to System RAM Consumption 4042
    - Measuring System RAM Consumption: Tools 4043
    - Measuring System RAM Consumption: Runtime 4059
    - Learn To Let Go (Of Your Heap) 4060
- Issues with Battery Life
    - Prerequisites 4063
    - You're Getting Blamed 4064
    - Not All Batteries Are Created Equal 4065
    - Stretching Out the Last mWh 4065
- Power Measurement Options
    - Prerequisites 4067
    - batterystats and the Battery Historian 4068
    - PowerTutor 4079
    - Battery Screen in Settings Application 4083
    - BatteryInfo Dump 4085
- Sources of Power Drain
    - Prerequisites 4089
    - Screen 4090
    - Disk I/O 4091
    - WiFi and Mobile Data 4092
    - GPS 4095
    - Camera 4096
    - Additional Sources 4096
- Addressing Application Size Issues
    - Prerequisites 4099
    - The APK Analyzer 4099
    - Java Code, and the 64K Method Limit 4102
    - Native Code 4107
    - Images 4108

    - APK Expansion Files 4110
- Crash Reporting Using ACRA
    - Prerequisites 4111
    - What Happens When Things Go “Boom”? 4111
    - Introducing ACRA 4112
    - Where ACRA Reports Crashes 4113
    - ACRA Integration Basics 4115
    - What the User Sees 4120
    - What You See 4127
    - Customizing Where Reports Go 4134
    - Adding Additional Data 4135
    - Removing Data 4138
    - End-User Configuration 4139
    - ACRA and Processes 4139
- In-App Diagnostics
    - Prerequisites 4143
    - The Diagnostic Activity 4144
    - The Diagnostic Web App 4155
    - The Diagnostic Overlay 4163
- Anti-Patterns
    - Prerequisites 4179
    - Leak Threads... Or Things Attached to Threads 4179
    - Use Large Heap Unnecessarily 4181
    - Misuse the MENU Button 4183
    - Interfere with Navigation 4184
    - Use android:sharedUserId 4186
    - Implement a “Quit” Button 4187
    - Terminate Your Process 4189
    - Try to Hide from the User 4190
    - Use Multiple Processes 4191
    - Hog System Resources 4193
- Widget Catalog: AdapterViewFlipper
    - Key Usage Tips 4197
    - A Sample Usage 4198
    - Visual Representation 4198
- Widget Catalog: CalendarView
    - Key Usage Tips 4199
    - A Sample Usage 4200
    - Visual Representation 4201
- Widget Catalog: DatePicker
    - Key Usage Tips 4205

    - A Sample Usage 4206
    - Visual Representation 4208
- Widget Catalog: ExpandableListView
    - Key Usage Tips 4213
    - A Sample Usage 4214
    - Visual Representation 4220
- Widget Catalog: SeekBar
    - Key Usage Tips 4223
    - A Sample Usage 4223
    - Visual Representation 4226
- Widget Catalog: SlidingDrawer
    - Key Usage Tips 4229
    - A Sample Usage 4230
    - Visual Representation 4231
- Widget Catalog: StackView
    - Key Usage Tips 4235
    - A Sample Usage 4236
    - Visual Representation 4237
- Widget Catalog: TabHost and TabWidget
    - Deprecation Notes 4239
    - Key Usage Tips 4239
    - A Sample Usage 4240
    - Visual Representation 4242
- Widget Catalog: TimePicker
    - Key Usage Tips 4245
    - A Sample Usage 4245
    - Visual Representation 4247
- Widget Catalog: ViewFlipper
    - Key Usage Tips 4251
    - A Sample Usage 4252
    - Visual Representation 4253
- Device Catalog: Chrome and Chrome OS
    - Prerequisites 4257
    - How This Works 4258
    - Testing Your App on Chrome OS 4259
    - Be Prepared To Be Wiped Out 4269
    - Enabling Your App for Chrome OS 4270
    - ***Your App on Chrome OS 4270***
    - Distribution Options 4279
    - Apps Sans Role 4279
    - Getting Help 4280

- Android Things Basics
  - Prerequisites 4281
  - Um, What's a Thing? 4282
  - Headless, But No Horseman 4283
  - Setting Up a Raspberry-Flavored Thing 4283
  - An App For Your Thing 4294
  - Control Interfaces 4298
  - What's Different? 4301
  - Environment Details 4304
  - Considerations 4306
- Device Catalog: BlackBerry
  - I Thought BlackBerry Had Their Own OS? 4310
  - What Else Is Different? 4310
  - What Are We Making? 4313
  - Getting Your Development Environment Established 4313
  - How Does Distribution Work? 4316
- Device Catalog: Android TV
  - Prerequisites 4319
  - Hey, Wait a Minute... I Thought the Name Was "Google TV"? 4319
  - Some Android TV Hardware 4320
  - What Features and Configurations Does It Use? 4322
  - What Is Really Different? 4323
  - Getting Your Development Environment Established 4325
  - How Does Distribution Work? 4327
- Device Catalog: Amazon Fire TV and Fire TV Stick
  - Prerequisites 4331
  - Introducing the Fire TV Devices 4331
  - What Features and Configurations Do They Use? 4337
  - What Is Really Different? 4338
  - Casting and Fire TV 4339
  - Getting Your Development Environment Established 4340
  - Working with the Remote and Controller 4341
  - How Does Distribution Work? 4343
  - Getting Help 4343
- Device Catalog: Samsung DeX
  - DeX Screen Modes 4347
  - Other App Impacts 4349
  - For More Information 4354
- Appendix A: CWAC Libraries
  - cwac-adapter 4357
  - cwac-colormixer 4357

- cwac-crossport 4358
- cwac-document 4358
- cwac-layouts 4358
- cwac-merge 4358
- cwac-netsecurity 4359
- cwac-presentation 4359
- cwac-provider 4359
- cwac-sacklist 4359
- cwac-saferoom 4359
- cwac-security 4360
- cwac-strictmodeex 4360
- cwac-wakeful 4360

Appendix B: Android 8.0

- ***The War on Background Processing, Continued 4361***
- ***JobScheduler Enhancements 4368***
- Auto-Fill 4369
- Notification Channels 4370
- Other Changes with Notifications 4370
- Multi-Window Changes 4371
- WebView Changes 4371
- ContentProvider Changes 4371
- Storage Access Framework Changes 4371
- Package Management 4372
- Fonts as Resources 4372
- Other Major Changes in Android 8.0 4373
- Other Minor Changes in Android 8.0 4381

Appendix C: Community Theater and the Appinars

- Viewing the Appinar Roster 4387
- Managing Appinars 4390
- Viewing an Appinar 4393

Preface

Welcome to the Book!

Thanks!

Thanks for your interest in developing applications for Android! Android has grown from nothing to arguably the world's most popular smartphone OS in a few short years. Whether you are developing applications for the public, for your business or organization, or are just experimenting on your own, I think you will find Android to be an exciting and challenging area for exploration.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

The Book's Structure

As you may have noticed, this is a rather large book.

To make this vast quantity of material manageable, the chapters are divided into the *core* chapters and a series of *trails*.

The core chapters represent many key concepts that Android developers need to understand in order to build an app. While an occasional "nice to have" topic will drift into the core — to help illustrate a point, for example — the core chapters generally are fairly essential.

The core chapters are designed to be read in sequence and will interleave both traditional technical book prose with tutorial chapters, to give you hands-on experience with the concepts being discussed. Most of the tutorials can be skipped,

though the first two — covering setting up your SDK environment and creating a project – everybody should read.

The bulk of the chapters are divided into trails, covering some particular general topic, from data storage to advanced UI effects to performance measurement and tuning. Each trail will have several chapters. However, those chapters, and the trails themselves, are not necessarily designed to be read in any order. Each chapter in the trails will point out prerequisite chapters or concepts that you will want to have covered in advance. Hence, these chapters are mostly reference material, for when you specifically want to learn something about a specific topic.

The core chapters will link to chapters in the trails, to show you where you can find material related to the chapter you just read. So between the book's table of contents, this preface, the search tool in your digital book reader, and the cross-chapter links, you should have plenty of ways of finding the material you want to read.

You are welcome to read the entire book front-to-back if you wish. The trails will appear after the core chapters. Those trails will be in a reasonably logical order, though you may have to hop around a bit to cover all of the prerequisites.

The Trails

Here is a list of all of the trails and the chapters that pertain to those trails, in order of appearance (except for those appearing in the list multiple times, where they span major categories):

Code Organization and Gradle

- Working with Library Projects
- Gradle and Tasks
- Gradle Build Variants
- Manifest Merger Rules
- Signing Your App
- Distribution
- Advanced Gradle for Android Tips

Testing

- Testing with JUnit4

- Testing with Espresso
- Testing with UIAutomator
- Measuring Test Coverage
- Unit Testing
- MonkeyRunner and the Test Monkey

Rx

- Java 8 Lambda Expressions
- Rx Basics

Advanced UI

- Notifications
- Advanced Notifications
- Multi-Window Support
- Advanced ConstraintLayout
- GridLayout
- Dialogs and DialogFragments
- Advanced ListViews
- Action Modes and Context Menus
- Other Advanced Action Bar Techniques
- Toolbar
- AppCompat: The Official Action Bar Backport
- RecyclerView
- Implementing a Navigation Drawer
- The Android Design Support Library
- Advanced Uses of WebView
- The Input Method Framework
- Fonts
- Rich Text
- Animators
- Legacy Animations
- Custom Drawables
- Mapping with Maps V2
- Crafting Your Own Views
- Advanced Preferences
- Custom Dialogs and Preferences
- Progress Indicators
- More Fun with Pagers

- Focus Management and Accessibility
- Miscellaneous UI Tricks
- Event Bus Alternatives
- Tasks
- The Assist API ("Now On Tap")
- The Auto-Fill API
- Data Binding
- Drag-and-Drop
- Keyboard and Mouse Input
- Viewing PDFs

Home Screen Effects

- Home Screen App Widgets
- Adapter-Based App Widgets

Security

- SSL
- NetCipher
- Encrypted Storage
- Advanced Permissions
- Restricted Profiles and UserManager
- Device Authentication
- Keys and the Keystore
- Miscellaneous Security Techniques

Data Storage and Retrieval

- Content Provider Theory
- Content Provider Implementation Patterns
- The Loader Framework
- The ContactsContract Provider
- The CalendarContract Provider
- The MediaStore Provider
- Consuming Documents
- Providing Documents
- Encrypted Storage
- Packaging and Distributing Data
- Advanced Database Techniques

- Data Backup

Advanced Network Topics

- SSL
- NetCipher
- Embedding a Web Server
- Miscellaneous Network Capabilities

Media

- Audio Playback
- Audio Recording
- Video Playback
- Using the Camera via 3rd-Party Apps
- Working Directly with the Camera
- The MediaStore Provider
- Media Routes
- Supporting External Displays
- Google Cast and ChromeCast
- The "10 Foot UI"
- Putting the TVs All Together: Decktastic
- Creating a MediaRouteProvider
- The Media Projection APIs

Hardware and System Services

- AlarmManager and the Scheduled Service Pattern
- PowerManager and WakeLocks
- JobScheduler
- Accessing Location-Based Services
- The Fused Location Provider
- Working with the Clipboard
- Telephony
- Working With SMS
- NFC
- Device Administration
- Basic Use of Sensors
- Printing and Document Generation
- Basic Bluetooth RFCOMM

- Dealing with Different Hardware

Integration and Introspection

- Writing and Using Parcelables
- Responding to URLs
- App Shortcuts
- PackageManager Tricks
- Remote Services and the Binding Pattern
- Advanced Manifest Tips
- Miscellaneous Integration Tips
- Replacing App Code Dynamically

Other Tools

- Android Studio Dialogs and Editors
- Advanced Emulator Capabilities
- Lint and the Support Annotations
- Inspecting Layouts
- Screenshots and Screencasts
- ADB Tips and Tricks
- Stetho
- Finding CPU Bottlenecks
- Finding Memory Leaks

Tuning Android Applications

- Issues with Speed
- Finding CPU Bottlenecks
- NDK
- Improving CPU Performance in Java
- Finding and Eliminating Jank
- Issues with Bandwidth
- Focus On: TrafficStats
- Measuring Bandwidth Consumption
- Being Smarter About Bandwidth
- Issues with Application Heap
- Finding Memory Leaks
- Issues with System RAM
- Issues with Battery Life

- Other Power Measurement Options
- Sources of Power Drain
- Addressing Application Size Issues

Miscellaneous Topics

- Crash Reporting with ACRA
- In-App Diagnostics
- Anti-Patterns

Widget Catalog

- AdapterViewFlipper
- CalendarView
- DatePicker
- ExpandableListView
- SeekBar
- SlidingDrawer
- StackView
- TabHost
- TimePicker
- ViewFlipper

Device Catalog

- Chrome and Chrome OS
- Android Things Basics
- BlackBerry
- Google TV
- Amazon Fire TV
- Samsung DeX

Appendices

- Appendix A: CWAC Libraries
- Appendix B: O Developer Preview
- Appendix C: Community Theater and the Appinars

About the Updates

This book is updated frequently, typically every 6-8 weeks.

Each release has notations to show what is new or changed compared with the immediately preceding release:

- The Table of Contents shows sections with changes in bold-italic font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

What's New in Version 8.10?

For those of you who have read previous editions of this book, here are some of the highlights of what is new in the prose in Version 8.10.

This update:

- Adds a new chapter on authenticating the user against the device, including through fingerprints
- Adds a new chapter on working with the Android keystore, particularly for encrypting data with keys backed by device authentication, including through fingerprints
- Migrates some more material out of the Android 8.0 appendix into mainstream chapters; what remains will stick around as an appendix for the foreseeable future
- Begins a mass migration away from native fragments to the fragments backport, amid rumors that native fragments will be deprecated in Android P
- Drops the chapter on integrating a JVM-based scripting language, as the chapter was too old, and embedding a language interpreter raises security issues
- Drops the chapter on implementing a plugin-based system, as the chapter was too old (though this chapter may be revisited in the future)
- Fixes a variety of errata

Warescription

You (hopefully) are reading this digital book by means of a Warescription.

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats. You also have access to a version of the book as its own Android APK file, complete with high-speed full-text searching. You also have access to other titles that CommonsWare may publish during that subscription period.

Each subscriber gets personalized editions of all editions of each title. That way, your books are never out of date for long, and you can take advantage of new material as it is made available. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

However, you can only download the books while you have an active Warescription. There is a grace period after your Warescription ends: you can still download the book until the next book update comes out after your Warescription ends. After that, you can no longer download the book. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The CommonsWare Twitter feed
2. The CommonsBlog
3. The Warescription newsletter, which you can subscribe to off of your Warescription page
4. Just check back on the Warescription site every month or two

Subscribers also have access to other benefits, including:

- "Office hours" — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow "bump" service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- A discussion board for asking arbitrary questions about Android app development

About the APK Edition

In addition to classic digital book formats (PDF, EPUB, MOBI/Kindle), this book is available as an Android app, in the form of an APK file. This app has an integrated

digital book reader, showing you the same contents as you would find in the EPUB version of the book. However, it has a few features that are unique.

First, it has a very fast full-text-search index built in. You can quickly search for keywords, class names, and the like, with sub-second response time on most Android hardware. You can even use boolean search clauses (e.g., search on `encryption OR decryption`).

Second, it has Community Theater, where you can view appinars, or app-based training modules. These are presentations, complete with slides, videos, screencasts, source code, and more. Through Community Theater, you can view available appinars, download those of interest, and watch them when you want.

The APK edition of the book reader works on Android 4.0.3 and higher, though the Community Theater portion only works on Android 4.4 and higher.

Installation instructions for the APK edition can be found on the CommonsWare Web site. Details about using Community Theater can be found in an appendix of this book.

Book Bug Bounty

Find a problem in the book? Let CommonsWare know!

Be the first to report a unique concrete problem in the current digital edition, and CommonsWare will extend your Warescription by six months as a bounty for helping CommonsWare deliver a better product.

By "concrete" problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book
3. Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Be sure to check the book's errata page, though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about "softer" issues as well, such as:

1. Places where you think we are in error, but where we feel our interpretation is reasonable
2. Places where you think we could add sample applications, or expand upon the existing material
3. Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

The Book Bug Bounty also extends to the appinars that you view in the Community Theater portion of the APK edition of the book. Typos and similar concrete issues in an appinar will qualify. Be sure to point out which appinar it is and what slide (or code, screenshot, video, etc.) has the problem.

In addition, the Book Bug Bounty covers reproducible bugs in the APK itself. If you are having problems using the APK, due to crashes or some other problem, be sure to let us know, with sufficient steps to reproduce the problem (including information about the device that you are using, such as the Android OS version).

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code and Its License

The source code samples shown in this book are available for download from the book's GitHub repository. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

If you wish to use the source code from the GitHub repository, please follow the instructions on that repository's home page for details of how to use the projects in various development environments, notably Android Studio.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 February 2022*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit the Creative Commons Web site

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

I would like to thank the Android team, not only for putting out a good product, but for invaluable assistance on the Android Google Groups and Stack Overflow.

I would also like to thank the thousands of readers of past editions of this book, for their feedback, bug reports, and overall support.

Of course, thanks are also out to the overall Android ecosystem, particularly those developers contributing their skills to publish libraries, write blog posts, answer support questions, and otherwise contribute to the strength of Android.

Portions of this book are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Core Chapters

Key Android Concepts

No doubt, you are in a hurry to get started with Android application development. After all, you are reading this book, aimed at busy coders.

However, before we dive into getting tools set up and starting in on actual programming, it is important that we "get on the same page" with respect to several high-level Android concepts. This will simplify further discussions later in the book.

Android Applications

This book is focused on writing Android applications. An application is something that a user might install from the Play Store or otherwise download to their device. That application should have some user interface, and it might have other code designed to work in the background (multi-tasking).

This book is not focused on modifications to the Android firmware, such as writing device drivers. For that, you will need to seek other resources.

This book assumes that you have some hands-on experience with Android devices, and therefore you are familiar with buttons like HOME and BACK, the built-in Settings application, the concept of a home screen and launcher, and so forth. If you have never used an Android device, you are strongly encouraged to get one (new or used) and spend some time with it before starting in on learning Android application development.

Programming Language

The vast majority of Android applications are written exclusively in Java. Hence, that is what this book will spend most of its time on and will demonstrate with a seemingly infinite number of examples.

However, there are other options:

- You can write parts of the app in C/C++, for performance gains, porting over existing code bases, etc.
- You can write an entire app in C/C++, mostly for games using OpenGL for 3D animations
- You can write the guts of an app in HTML, CSS, and JavaScript, using tools to package that material into an Android application that can be distributed through the Play Store and similar venues
- And so on

Some of this will be covered later in the book, but the vast majority of this book is focused on Java-based app development.

The author assumes that you know Java at this point. If you do not, you will need to learn Java before you go much further. You do not need to know *everything* about Java, as Java is vast. Rather, focus on:

- Language fundamentals (flow control, etc.)
- Classes and objects
- Methods and data members
- Public, private, and protected
- Static and instance scope
- Exceptions
- Threads
- Collections
- Generics
- File I/O
- Reflection
- Interfaces

The links are to Wikibooks material on those topics, though there are countless other Java resources for you to consider.

Google also supports Kotlin as a programming language for Android apps. Kotlin can be used as a replacement for Java. While Kotlin has plenty of benefits, it is also a relatively new programming language. This book is focused on the Android SDK, more so than on Java (or Kotlin), and so regardless of which of those two languages you use, the material in this book will be relevant. This book's examples are mostly in Java, due to that being the dominant Android app development language today.

Components

When you first learned Java — whether that was yesterday or back when dinosaurs roamed the Earth — you probably started off with something like this:

```
class SillyApp {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

In other words, the entry point into your application was a `public static void` method named `main()` that took a `String` array of arguments. From there, you were responsible for doing whatever was necessary.

However, there are other patterns used elsewhere in Java. For example, you do not usually write a `main()` method when writing a Java servlet. Instead, you extend a particular class supplied by a framework (e.g., `HttpServlet`) to create a component, then write some metadata that enumerates your components and tell the framework when and how to use them (e.g., `WEB.XML`).

Android apps are closer in spirit to the servlet approach. You will not write a `public static void main()` method. Instead, you will create subclasses of some Android-supplied base classes that define various application components. In addition, you will create some metadata that tells Android about those subclasses.

There are four types of components, all of which will be covered extensively in this book:

Activities

The building block of the user interface is the *activity*. You can think of an activity as being the Android analogue for the window or dialog in a desktop application, or the page in a classic Web app. It represents a chunk of your user interface and, in

some cases, a discrete entry point into your app (i.e., a way for other apps to link to your app).

Normally, an activity will take up most of the screen, leaving space for some "chrome" bits like the clock, signal strength indicators, and so forth.

However, bear in mind that on some devices, the user will be able to work with more than one activity at a time, such as split-screen mode on a phone or tablet. So, while it is easy to think of activities as being equivalent to the screen, just remember that this is a simplification, and that reality is more complicated (as reality often is).

Services

Activities are short-lived and can be shut down at any time, such as when the user presses the BACK button. *Services*, on the other hand, are designed to keep running, if needed, independent of any activity, for a moderate period of time. You might use a service for checking for updates to an RSS feed, or to play back music even if the controlling activity is no longer operating. You will also use services for scheduled tasks (akin to Linux or macOS "cron jobs") and for exposing custom APIs to other applications on the device, though the latter is a relatively advanced capability.

Content Providers

Content providers provide a level of abstraction for any data stored on the device that is accessible by multiple applications. The Android development model encourages you to make your own data available to other applications, as well as your own — building a content provider lets you do that, while maintaining a degree of control over how your data gets accessed.

So, for example, if you have a PDF file that you downloaded on behalf of the user, and you want to allow the user to view that PDF file, you might create a content provider to make that PDF file available to other apps. You can then start up an activity that will be able to view that PDF, where Android and the user will determine what PDF-viewing activity handles that request.

Broadcast Receivers

The system, or applications, will send out *broadcasts* from time to time, for everything from the battery getting low, to when the screen turns off, to when

connectivity changes from WiFi to mobile data. A broadcast receiver can arrange to listen for these broadcasts and respond accordingly.

Widgets, Containers, and Resources

Most of the focus on Android application development is on the UI layer and activities. Most Android activities use what is known as "the widget framework" for rendering their user interface, though you are welcome to use the 2D (`Canvas`) and 3D (OpenGL) APIs as well for more specialized GUIs.

In Android terms, a *widget* is the "micro" unit of user interface. Fields, buttons, labels, lists, and so on are all widgets. Your activity's UI, therefore, is made up of one or more of these widgets. This is a common approach in most UI toolkits, and so most likely you have already worked with buttons, labels, fields, and similar sorts of widgets somewhere else previously.

If you have more than one widget — which is fairly typical — you will need to tell Android how those widgets are organized on the screen. To do that, you will use various container classes referred to as *layout managers*. These will let you put things in rows, columns, or more complex arrangements as needed.

To describe how the containers and widgets are connected, you will typically create a *layout resource file*. *Resources* in Android refer to things like images, strings, and other material that your application uses but is not in the form of some programming language source code. UI layouts are another type of resource. You will create these layouts either using a structured tool, such as an IDE's drag-and-drop GUI builder, or by hand in XML form.

Sometimes, your UI will work across all sorts of devices: phones, tablets, televisions, etc. Sometimes, your UI will need to be tailored for different environments. You will be able to put resources into *resource sets* that indicate under what circumstances those resources can be used (e.g., use these for normal-sized screens, but use those for larger screens).

We will be examining all of these concepts, in much greater detail, as we get deeper into the book.

Apps and Packages

Given a bucket of source code and a basket of resources, the Android build tools will give you an application as a result. The application comes in the form of an *APK file*. It is that APK file that you will upload to the Play Store or distribute by other means.

Each Android application has a package name, also referred to as an application ID. A package name must fulfill three requirements:

1. It must be a valid Java package name, as some Java source code will be generated by the Android build tools in this package
2. No two applications can exist on a device at the same time with the same application ID
3. No two applications can be uploaded to the Play Store having the same application ID

When you create your Android project — the repository of that source code and those resources — you will declare what package name is to be used for your app. Typically, you will pick a package name following the Java package name "reverse domain name" convention (e.g., `com.commonsware.android.foo`). That way, the domain name system ensures that your package name prefix (`com.commonsware`) is unique, and it is up to you to ensure that the rest of the package name distinguishes one of your apps from any other.

Android Devices

There are well in excess of one billion Android devices in use today, representing thousands of different models from dozens of different manufacturers. Android itself has evolved since Android 1.0 in 2008. Between different device types and different Android versions, many a media pundit has lobbed the term "fragmentation" at Android, suggesting that creating apps that run on all these different environments is impossible.

In reality, it is not that bad. Some apps will have substantial trouble, but most apps will work just fine if you follow the guidance presented in this book and in other resources.

Types

Android devices come in all shapes, sizes, and colors. However, there are three dominant "form factors":

- the phone
- the tablet
- the notebook (mostly, these run Chrome OS, which has some support for Android apps)

Beyond that, there are several less-common form factors

- the television (TV)
- the wearable (smart watches, etc.)
- the desktop (a device designed to be plugged into a monitor, keyboard, and mouse)

You will often hear developers and pundits refer to these form factors, and this book will do so from time to time as well. However, it is important that you understand that Android has no built-in concept of a device being a "phone" or a "tablet" or a "TV". Rather, Android distinguishes devices based on capabilities and features. So, you will not see an `isPhone()` method anywhere, though you can ask Android:

- what is the screen size?
- does the device have telephony capability?
- etc.

Similarly, as you build your applications, rather than thinking of those form factors, focus on what capabilities and features you need. Not only will this help you line up better with how Android wants you to build your apps, but it will make it easier for you to adapt to other form factors that will come about such as:

- airplane seat-back entertainment centers
- in-car navigation and entertainment devices
- and so on

The Emulator

While there are ~2 billion Android devices representing ~10,000 device models, probably you do not have one of each model. You may only have a single piece of

Android hardware. And if you do not even have that, you most certainly will want to acquire one before trying to publish an Android app.

To help fill in the gaps between the devices you have and the devices that are possible, the Android developer tools ship an *emulator*. The emulator behaves like a piece of Android hardware, but it is a program you run on your development machine. You can use this emulator to emulate many different devices, with different screen sizes and Android OS versions, by creating one or more Android virtual devices, or *AVDs*.

In an upcoming chapter, we will discuss how you install the Android developer tools and how you will be able to create these AVDs and run the emulator.

OS Versions and API Levels

Android has come a long way since the early beta releases from late 2007. Each new Android OS version adds more capabilities to the platform and more things that developers can do to exploit those capabilities.

Moreover, the core Android development team tries very hard to ensure forwards and backwards compatibility. An app you write today should work unchanged on future versions of Android (forwards compatibility), albeit perhaps missing some features or working in some sort of "compatibility mode". And there are well-trod paths for how to create apps that will work both on the latest and on previous versions of Android (backwards compatibility).

To help us keep track of all the different OS versions that matter to us as developers, Android has *API levels*. A new API level is defined when an Android version ships that contains changes that affect developers. When you create an emulator AVD to test your app, you will indicate what API level that emulator should emulate. When you distribute your app, you will indicate the oldest API level your app supports, so the app is not installed on older devices.

At the time of this writing, the API levels of significance to most Android developers are:

- API Level 19 (Android 4.4)
- API Level 21 (Android 5.0)
- API Level 22 (Android 5.1)
- API Level 23 (Android 6.0)
- API Level 24 (Android 7.0)

- API Level 25 (Android 7.1)

Here, "of significance" refers to API levels that have a reasonable number of Android devices — 5% or more, as reported by the "Platform Versions" dashboard chart.

The latest version of Android is 8.1, API Level 27.

Note that API Level 20 was used for the version of Android 4.4 running on the first-generation Android Wear devices. Unless you are specifically developing apps for Wear, you will not be worrying much about API Level 20.

Dalvik and ART

In terms of Android, Dalvik and ART are virtual machines (VM)s. Virtual machines are used by many programming languages, such as Java, Perl, and Smalltalk. Dalvik and ART are designed to work much like a Java VM, but optimized for embedded Linux environments.

Primarily, the difference between the two is that ART is used on Android 5.0 and higher, while Dalvik was used on older devices. In truth, the story is more complicated than this, but this will do for now.

So, what really goes on when somebody writes an Android application is:

1. Developers write Java-syntax source code, leveraging class libraries published by the Android project and third parties.
2. Developers compile the source code into Java VM bytecode, using the `javac` compiler that comes with the Java SDK.
3. Developers translate the Java VM bytecode into Dalvik VM bytecode, which is packaged with other files into a ZIP archive with the `.apk` extension (the APK file).
4. An Android device or emulator runs the APK file, causing the bytecode to be executed by an instance of a Dalvik or ART VM.

From your standpoint, most of this is hidden by the build tools. You pour Java source code into the top, and the APK file comes out the bottom.

However, there will be places from time to time where the differences between the Dalvik VM and the traditional Java VM will affect application developers, and this book will point out some of them where relevant.

Processes and Threads

When your application runs, it will do so in its own process. This is not significantly different than any other traditional operating system. Part of Dalvik's magic is making it possible for many processes to be running many Android applications at one time without consuming ridiculous amounts of RAM.

Android will also set up a batch of threads for running your app. The thread that your code will be executed upon, most of the time, is variously called the "main application thread" or the "UI thread". You do not have to set it up, but, as we will see later in the book, you will need to pay attention to what you do and do not do on that thread. You are welcome to fork your own threads to do work, and that is fairly common, though in some places Android handles that for you behind the scenes.

Don't Be Scared

Yes, this chapter threw a lot of terms at you. We will be going into greater detail on all of them in this book. However, Android is like a jigsaw puzzle with lots of interlocking pieces. To be able to describe one concept in detail, we will need to at least reference some of the others. Hence, this chapter was meant to expose you to terms, in hopes that they will sound vaguely familiar as we dive into the details.

Choosing Your Development Toolchain

Before you go much further in your Android endeavors (or possibly endeavours, depending upon your preferred spelling), you will need to determine what toolchain you will use to build your Android applications.

Android Studio

The current Google-backed Android IDE is Android Studio. Based on IntelliJ IDEA, Android Studio is the new foundation of Google's efforts to give Android developers top-notch development tools.

The next chapter contains a section with instructions on how to set up Android Studio.

Note, though, that Android Studio requires a fairly powerful development machine to work well: a fast CPU, lots of RAM (8GB or more), and an SSD are all strongly recommended.

Eclipse

Eclipse is also a popular IDE, particularly for Java development. Eclipse was Google's original IDE for Android development, by means of the Android Developer Tools (ADT) add-in, which gives the core of Eclipse awareness of Android. The ADT add-in, in essence, takes regular Eclipse operations and extends them to work with Android projects.

Note, though, that Google has discontinued maintenance of ADT. The Eclipse Foundation is setting up the "Andmore" project to try to continue work on allowing Eclipse to build Android apps. This book does not cover the Andmore project at this time, and developers are **strongly** encouraged to not use the ADT-enabled Eclipse from Google.

IntelliJ IDEA

While Android Studio is based on IntelliJ IDEA, you can still use the original IntelliJ IDEA for Android app development. A large subset of the Android Studio capabilities are available in the Android plugin for IDEA. Plus, the commercial IDEA Ultimate Edition will go beyond Android Studio in many areas outside of Android development.

In particular, if you are looking for "the one true IDE" that you can use for Android and non-Android projects, you should consider IntelliJ IDEA. Android Studio is nice, but it is mostly for Android projects.

Command-Line Builds via Gradle

And, of course, you do not need to use an IDE at all. While this may sound sacrilegious to some, IDEs are not the only way to build applications. Much of what is accomplished via an IDE can be accomplished through command-line equivalents, meaning a shell and an editor is all you truly need.

The recommended way to build Android apps outside of an IDE is by means of Gradle. Google has published a Gradle plugin that teaches Gradle how to build Android apps. Android Studio itself uses Gradle for its builds, so a single build configuration (e.g., `build.gradle` files) can be used both from an IDE and from a build automation tool like a continuous integration server.

An upcoming chapter gets into more about what Gradle (and the Android Gradle Plugin) are all about.

Yet Other Alternatives

Other IDEs have their equivalents of the ADT, albeit with minimal assistance from Google. For example, NetBeans has support via the NBAndroid add-on, and reportedly this has advanced substantially in the past few years.

You will also hear reference to using Apache Ant for doing command-line builds of Android apps. This has been supplanted by the Android Gradle Plugin at this time, and there is little support for Apache Ant anymore. Newcomers to Android are encouraged to not invest time in new work with Apache Ant for Android development projects.

IDEs… And This Book

You are encouraged to use Android Studio as you work through this book. You are welcome to use another IDE if you wish. You are even welcome to skip the IDE outright and just use an editor.

This book is focused primarily on demonstrating Android capabilities and the APIs for exploiting those capabilities. Hence, the sample code will work with any IDE. However, this book will cover some Android Studio-specific instructions, since that is the predominant Android IDE in use today.

What We Are Not Covering

In the beginning (a.k.a., 2007), we were lucky to have any means of creating an Android app.

Nowadays, there seems to be no end to the means by which we can create an Android app.

There are a few of these "means", though, that are specifically out of scope for this book.

App Inventor

You may also have heard of a tool named App Inventor and wonder where it fits in with all of this.

App Inventor was originally created by an education group within Google, as a means of teaching students how to think about programming constructs (branches, loops, etc.) and create interesting output (Android apps) without classic programming in Java or other syntax-based languages. App Inventor is purely drag-and-drop, both of widgets *and application logic*, the latter by means of "blocks" that snap together to form logic chains.

App Inventor was donated by Google to MIT, which has recently re-opened it to the public.

However, App Inventor is a closed system — at the present time, it does not somehow generate Java code that you can later augment. That limits you to whatever App Inventor is natively capable of doing, which, while impressive in its own right, offers a small portion of the total Android SDK capabilities.

App Generators

There are a seemingly infinite number of "app generators" available as online services. These are designed mostly for creating apps for specific vertical markets, such as apps for restaurants or apps for grocers. The resulting apps are mostly "brochure-ware", with few capabilities beyond a mobile Web site, yet still requiring the user to find, download, and install the app. Few of these generators provide the source code to the generated app, to allow the apps to be customized beyond what the generator generates.

Tutorial #1 - Installing the Tools

Now, let us get you set up with the pieces and parts necessary to build an Android app.

NOTE: The instructions presented here are accurate as of the time of this writing. However, the tools change rapidly, and so these instructions may be out of date by the time you read this. Please refer to the Android Developers Web site for current instructions, using this as a base guideline of what to expect.

But First, Some Notes About Android's Emulator

The Android tools include an emulator, a piece of software that pretends to be an Android device. This is very useful for development — not only does it mean you can get started on Android without a device, but the emulator can help test device configurations that you do not own.

There are two types of emulator: x86 and ARM. These are the two major types of CPUs used for Android devices. You *really* want to be able to use the x86 emulator, as the ARM emulator is extremely slow.

However, to use the x86 emulator, your development machine must have things set up properly first. Linux users need KVM, while Mac and Windows users need the "Intel Hardware Accelerated Execution Manager" (a.k.a., HAXM).

Also, this only works for certain CPU architectures, ones that support virtualization in hardware:

- Intel Virtualization Technology (VT, VT-x, vmx) extensions
- AMD Virtualization (AMD-V, SVM) extensions (Linux only)

Those virtualization extensions must also be enabled in your device's BIOS, and other OS-specific modifications may be required.

Also, at least for newer API levels, your CPU must support SSSE3 extensions, though the details of this requirement are not documented as of October 2017.

Part of the Android Studio installation process will try to set you up to be able to use the x86 emulator. Make note of any messages that you see in the installation wizard regarding "HAXM" (or, if you are running Linux, KVM), as those will be important later.

Step #1: Checking Your Hardware

Compiling and building an Android application, on its own, can be a hardware-intensive process, particularly for larger projects. Beyond that, your IDE and the Android emulator will stress your development machine further. Of the two, the emulator poses the bigger problem.

The more RAM you have, the better. 8GB or higher is a very good idea if you intend to use an IDE and the emulator together. If you can get an SSD for your data storage, instead of a conventional hard drive, that too can dramatically improve the IDE performance.

A faster CPU is also a good idea. The Android SDK emulator, as of 2016, supports CPUs with multiple cores — previously, it only supported a single core. However, other processes on your development machine will be competing with the emulator for CPU time, and so the faster your CPU is, the better off you will be. Ideally, your CPU has 2 to 4 cores, each 2.5GHz or faster at their base speed.

Beyond that, to use the x86 emulator — the emulator that runs well — you need a CPU with certain features:

Development OS	CPU Requirements
Windows	an Intel CPU with support for VT-x, EM64T, and "Execute Disable" (XD)
Mac	any
Linux	an Intel CPU with support for VT-x, EM64T, and "Execute Disable" (XD), or an AMD CPU with support for AMD-V

If your CPU does not meet those requirements, you will want to have 1+ Android devices available to you, so that you can test on hardware.

Also, if you are running Windows or Linux, you need to ensure that your computer's BIOS is set up to support Intel's virtualization extensions. Unfortunately, many PC manufacturers disable this by default. The details of how to get into your BIOS settings will vary by PC, but usually it involves rebooting your computer and pressing some function key on the initial boot screen. In the BIOS settings, you are looking for references to "virtualization" or "VT-x". Enable them if they are not already enabled. macOS machines come with virtualization extensions pre-enabled.

Step #2: Setting Up Java and 32-Bit Linux Support

When you write Android applications, you typically write them in Java source code. That Java source code is then turned into the stuff that Android actually runs (Dalvik bytecode in an APK file).

Android Studio, starting with version 2.2, ships with its own private copy of OpenJDK 8, and it will use that by default. You are welcome to install and use your own JDK if you wish, though ideally it is for Java 8.

If your development OS is Linux, make sure that you can run 32-bit Linux binaries. This may or may not already be enabled in your Linux distro. For example, on Ubuntu 17.10, you may need to run the following to get the 32-bit binary support installed that is needed by the Android build tools:

```
sudo apt-get install libncurses5:i386 libstdc++6:i386 zlib1g:i386
```

Step #3: Install Android Studio

As noted in the previous chapter, there are a few developer tools that you can choose from.

This book's tutorials focus on Android Studio. You are welcome to attempt to use Eclipse, another IDE, or no IDE at all for building Android apps. However, you will need to translate some of the tutorials' IDE-specific instructions to be whatever is needed for your development toolchain of choice.

At the time of this writing, the current production version of Android Studio is 3.0.1, and this book covers that version. If you are reading this in the future, you

may be on a newer version of Android Studio, and there may be some differences between what you have and what is presented here.

You have two major download options. You can get the latest shipping version of Android Studio from the Android Studio download page.

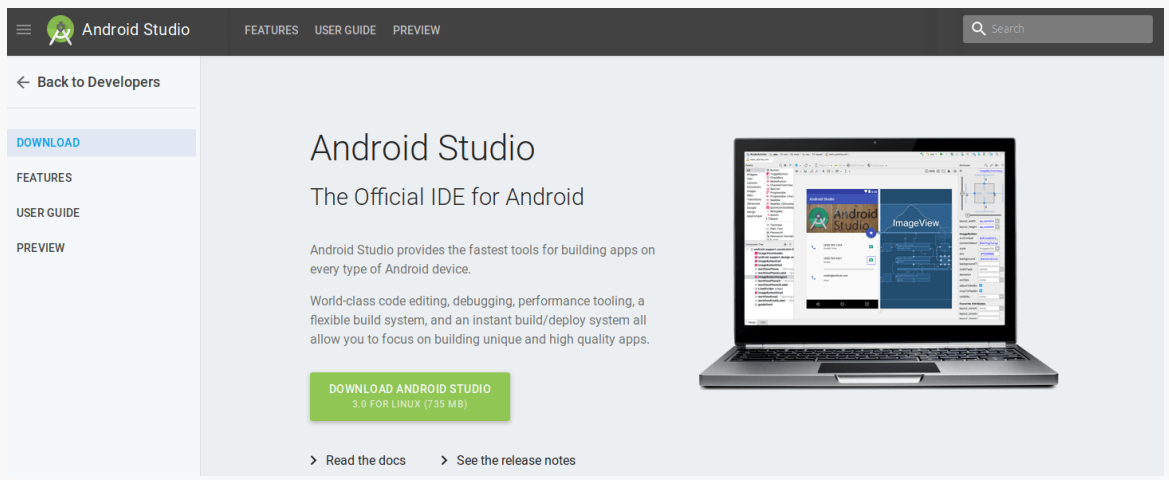


Figure 1: Android Studio Download Page

Or, you can download Android Studio 3.0.1 — the version used in this edition of this book — directly, for:

- Windows
- macOS
- Linux

Windows users can download a self-installing EXE, which will add suitable launch options for you to be able to start the IDE.

macOS users can download a DMG disk image and install it akin to other macOS software, dragging the Android Studio icon into the Applications folder.

Linux users (and power Windows users) can download a ZIP file, then unZIP it to some likely spot on your hard drive. Android Studio can then be run from the `studio` batch file or shell script from your Android Studio installation's `bin/` directory.

Step #4: Install the SDKs and Add-Ons

Next, we need to review what pieces of the Android SDK we have already and perhaps install some new items. To do that, you need to access the SDK Manager.

When you first run Android Studio, you may be asked if you want to import settings from some other prior installation of Android Studio:

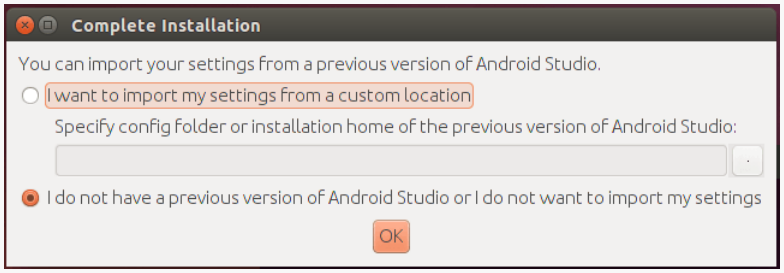


Figure 2: Android Studio First-Run Settings Migration Dialog

For most users, particularly those using Android Studio for the first time, the "I do not have..." option is the correct choice to make.

Then, after a short splash screen and a potentially long "Finding Available SDK Components" progress dialog, you will be taken to the Android Studio Setup Wizard:

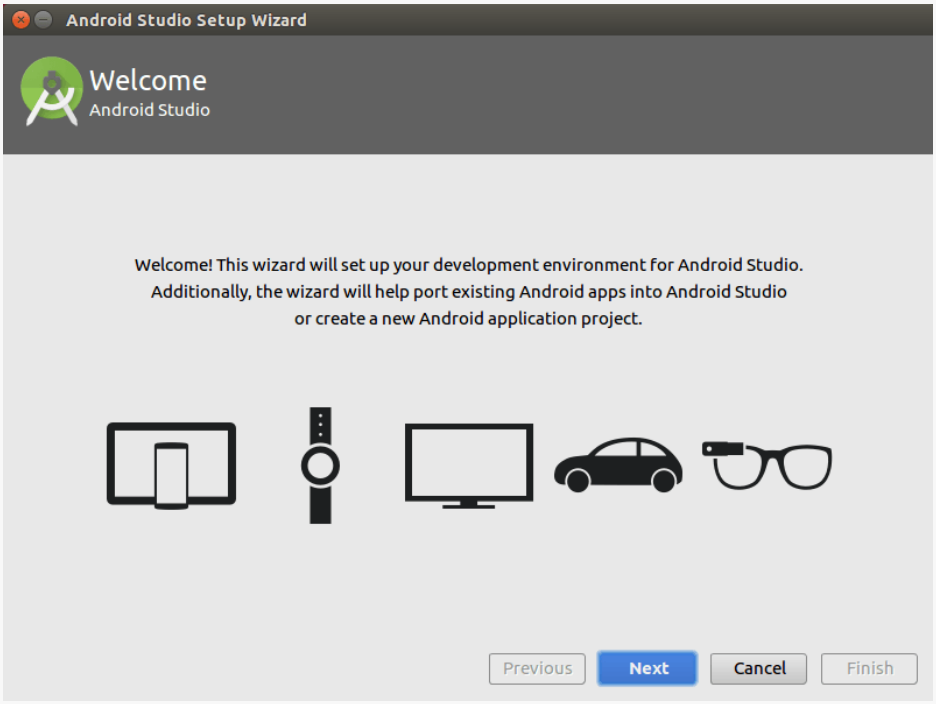


Figure 3: Android Studio Setup Wizard, First Page

Just click "Next" to advance to the second page of the wizard:

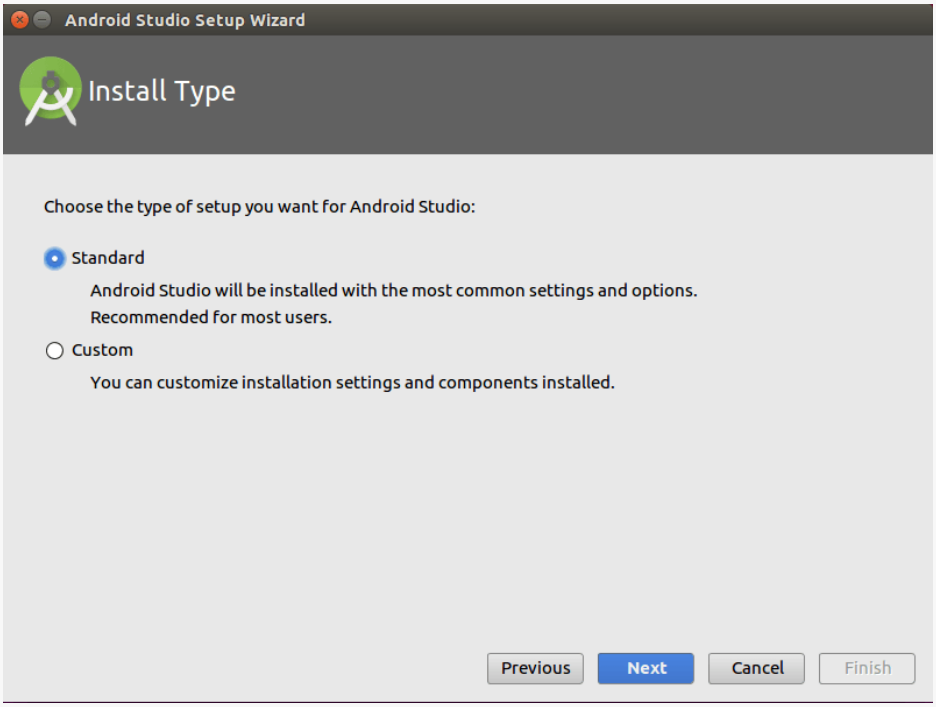


Figure 4: Android Studio Setup Wizard, Second Page

Here, you have a choice between “Standard” and “Custom” setup modes. Most likely, right now, the “Standard” route will be fine for your environment.

If you go the “Standard” route and click “Next”, you should be taken to a wizard page where you can choose your UI theme:

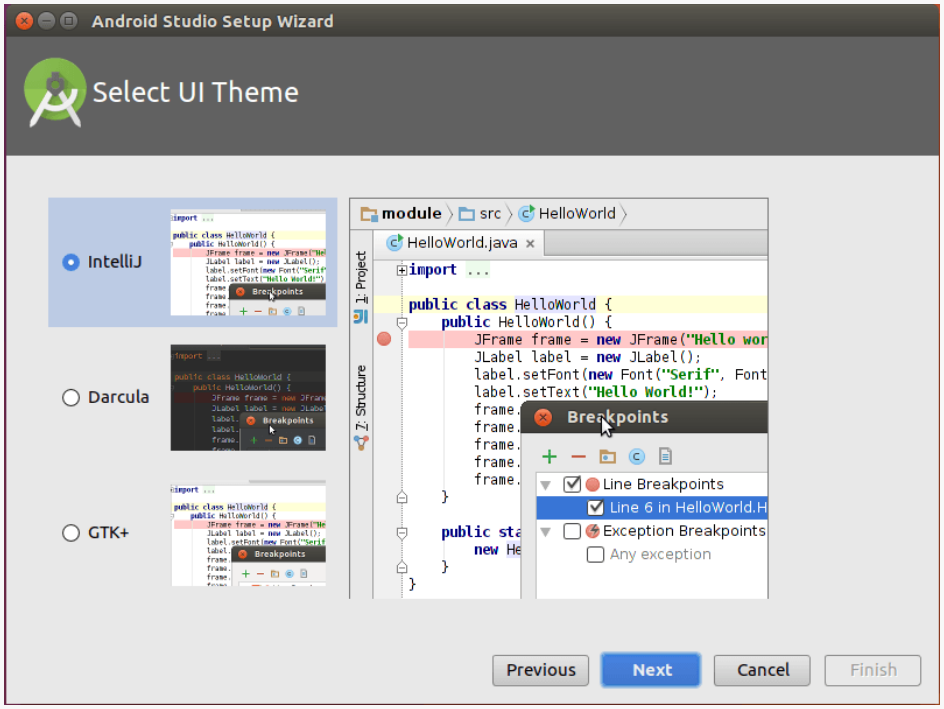


Figure 5: Android Studio Setup Wizard, UI Theme Page

Choose whichever you like, then click Next, to go to a wizard page to verify what will be downloaded and installed:

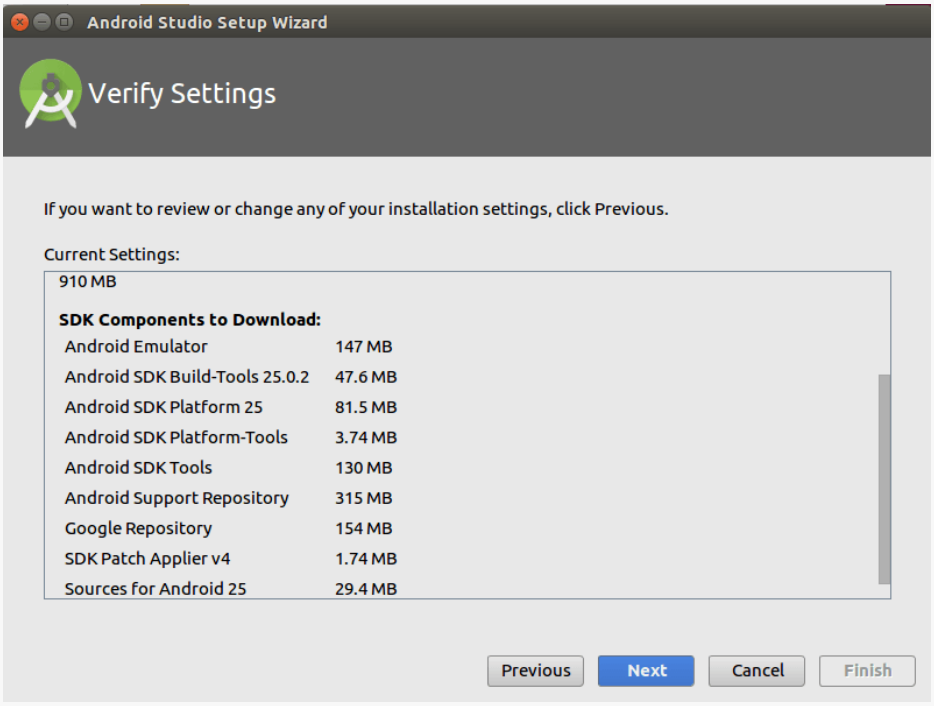


Figure 6: Android Studio Setup Wizard, Verify Settings Page

Clicking Next may take you to a wizard page explaining some information about the Android emulator:

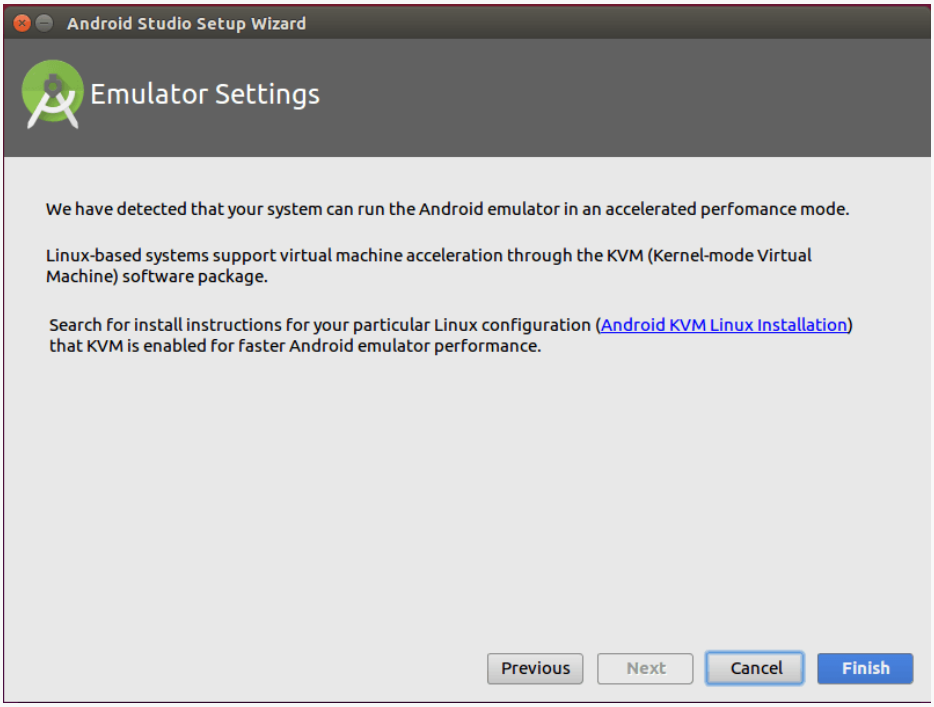


Figure 7: Android Studio Setup Wizard, Emulator Info Page

What is explained on this page may not make much sense to you. That is perfectly normal, and we will get into what this page is trying to say later in the book. Just click "Finish" to begin the setup process. This will include downloading a copy of the Android SDK and installing it into a directory adjacent to where Android Studio itself is installed.

When that is done, after clicking "Finish", Android Studio will busily start downloading stuff to your development machine. If you are running Linux, and your installation crashes with an "Unable to run mksdcard SDK tool" error, go back to Step #2 and set up 32-bit support on your Linux environment.

Clicking "Finish" will then take you to the Android Studio Welcome dialog:

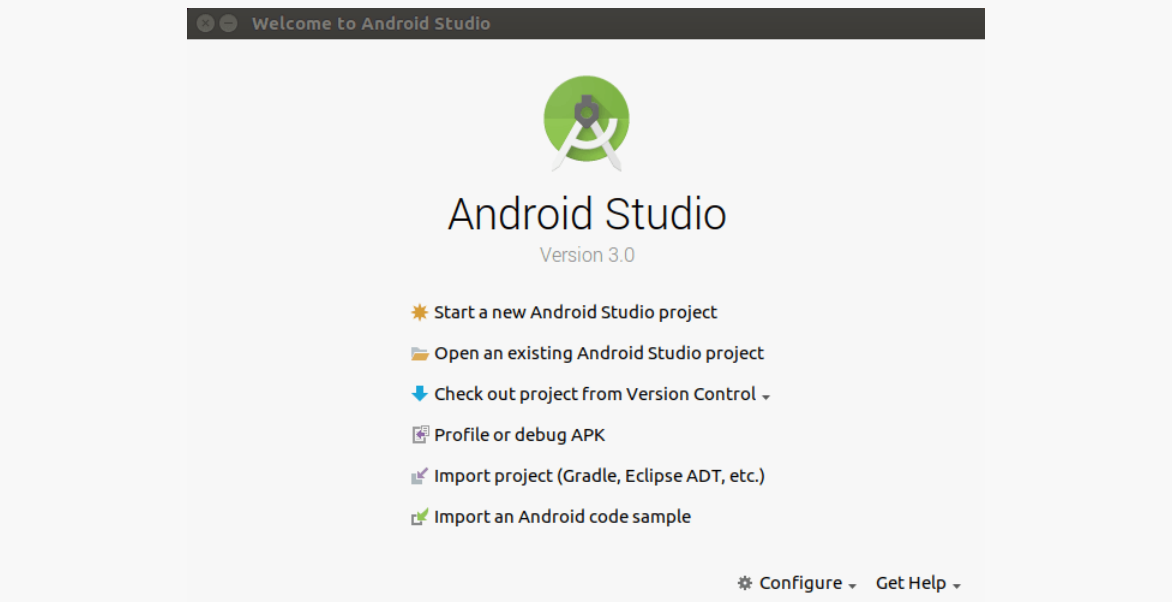


Figure 8: Android Studio Welcome Dialog

Then, in the welcome dialog, click Configure, to bring up a configuration drop-down list:

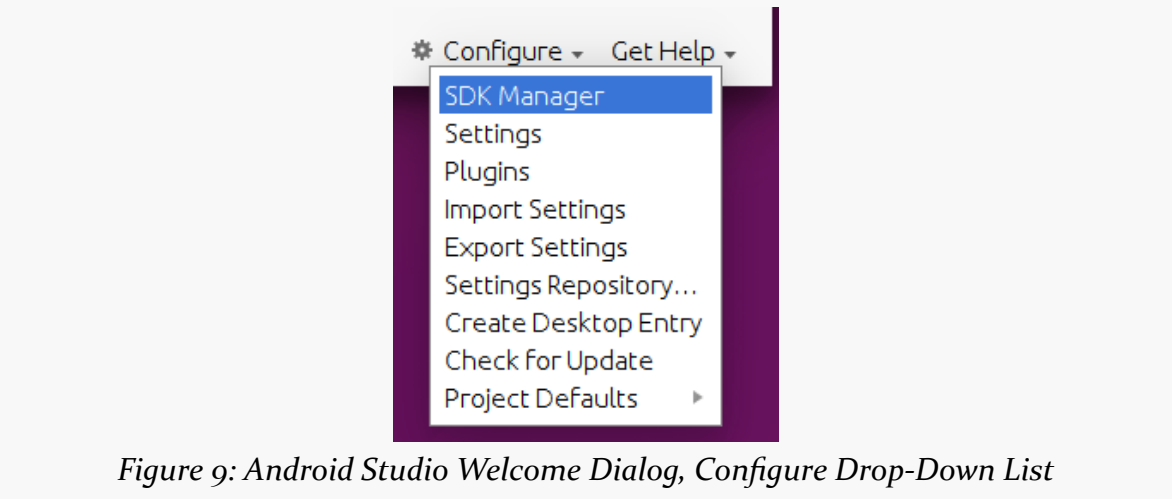


Figure 9: Android Studio Welcome Dialog, Configure Drop-Down List

There, tap on SDK Manager to bring up the SDK Manager.

Using SDK Manager and Updating Your Environment

You should now have the SDK Manager open, as part of the overall default settings for Android Studio:

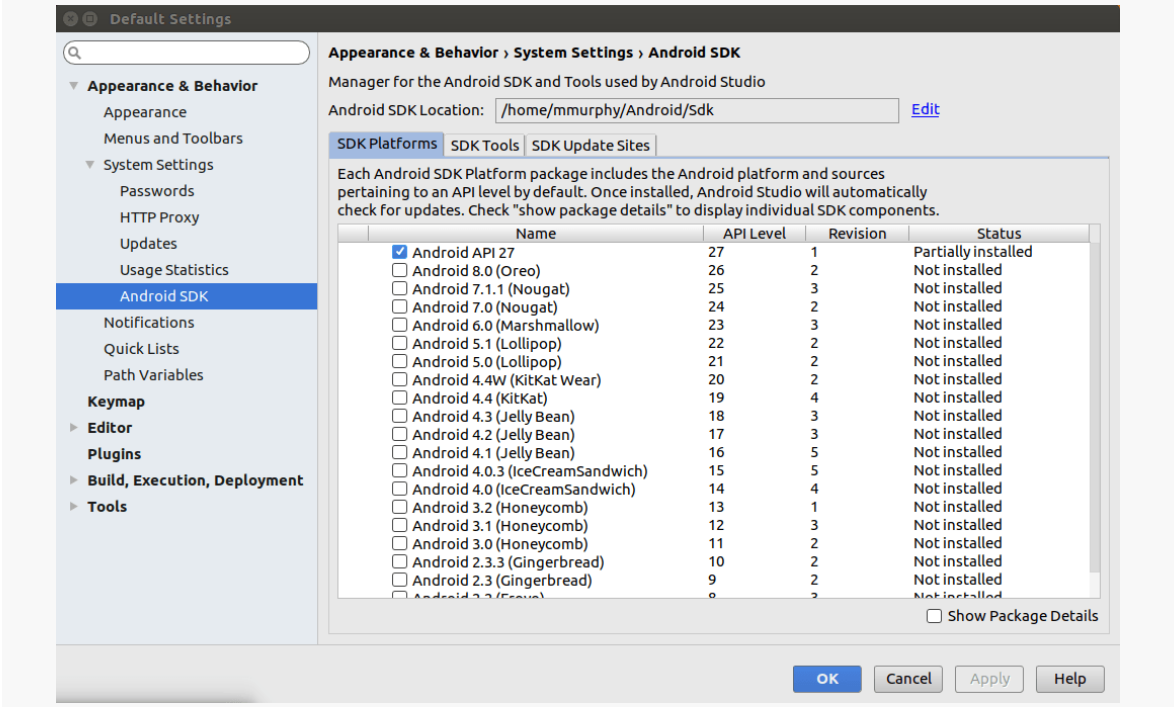


Figure 10: Android SDK Manager, "SDK Platforms" Tab

The "SDK Platforms" tab lists the versions of Android that you can compile against. The latest version of Android (or possibly a preview edition) is usually installed when you set up Android Studio initially. However, for the tutorials, please also check "Android 7.1 (Nougat)" in the list if it is not already checked, and then click the "Apply" button to download and install those versions. You may need to accept a license confirmation dialog as part of this process — review the license, and if you accept it, click Next to begin the download:

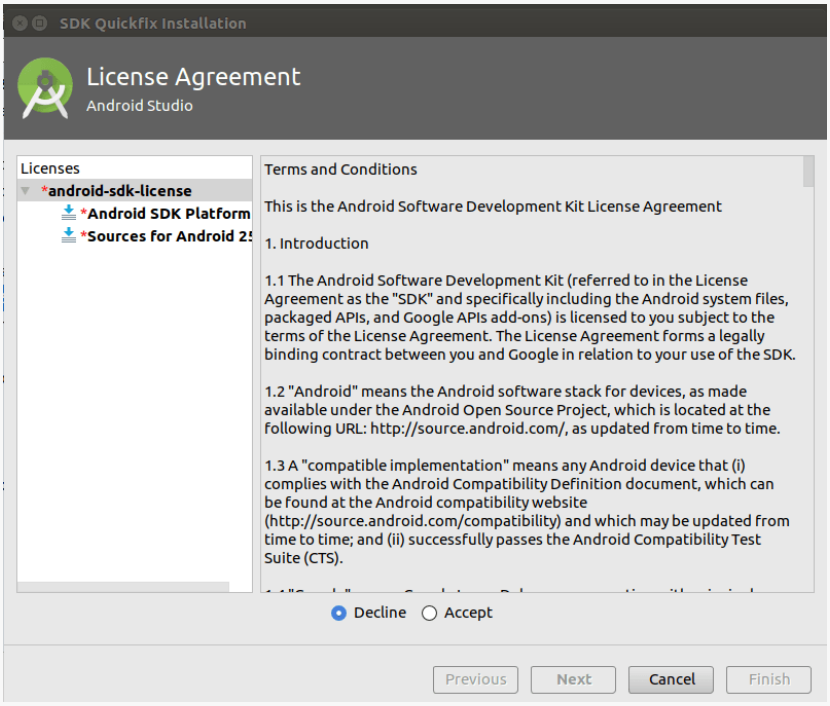


Figure 11: Android SDK Manager, License Confirmation Dialog

When that has completed, you can click "Finish" to close up the download dialog, and then you will be returned to the SDK Manager. Then, you can close up the SDK Manager by clicking the OK button.

In Our Next Episode…

… we will create an Android project that will serve as the basis for all our future tutorials, plus set up our emulator and device.

Android and Projects

When you work on creating an app for Android, you will do so by working in a "project". The project is a directory containing your source code and other files, like images and UI definitions. Your IDE or other build tools will take what is in your project and generate an Android app (APK) as output.

The details of how you get started with a project vary based upon what IDE you are using, so this chapter goes through the various possibilities.

Projects and Android Studio

With Android Studio, to work on a project, you can either create a new project from scratch, you can copy an existing Android Studio project to a new one, or you can import an existing Android project into Android Studio. The following sections will review the steps needed for each of these.

Creating a New Project

You can create a project from one of two places:

- If you are at the initial dialog that you first encountered when you opened Android Studio, choose the "Start a new Android Studio project..." menu item
- If you are inside the Android Studio IDE itself, choose File > New > New Project... from the main menu

This brings up the new-project wizard:

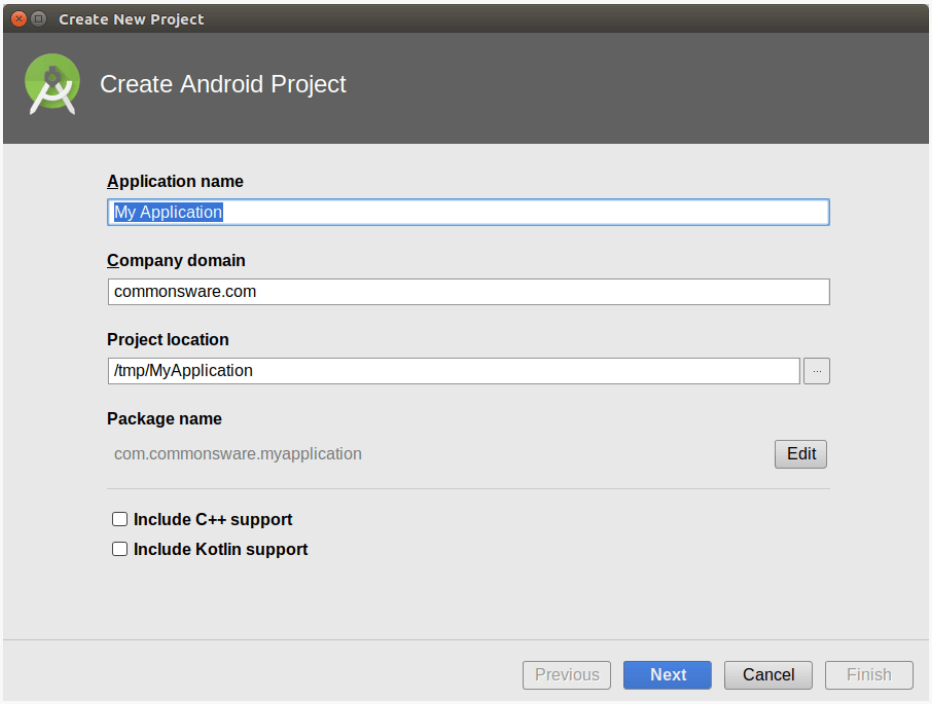


Figure 12: Android Studio Create-Project Wizard, First Page

The first page of the wizard is where you can specify:

- The application name, which is the initial name of your project as seen by the user, in places like your home screen launcher icon and the list of installed applications.
- The package name, which refers to a Java package name (e.g., `com.commonsware.empublite`). This package name will be used for generating some Java source code, and it also is used as a unique identifier of this app, as was mentioned earlier in this book.
- The directory where you want the project files to go
- Whether you know right now if you will be using C++ or not, as part of the Android Native Development Kit (NDK)
- Whether you know right now if you will be using Kotlin or not

Nothing that you choose here is permanent; you can revise everything later on if needed. The most painful to change is the package name, so ideally you choose a good value up front.

By default, the package name will be made up of two pieces:

1. The domain name that you specify in the “Company Domain” field
2. The application name, converted into all lowercase with no spaces or other punctuation

If this is not what you want, click the “Edit” button on the far right side of the proposed package name, which will now allow you to edit the package name directly:

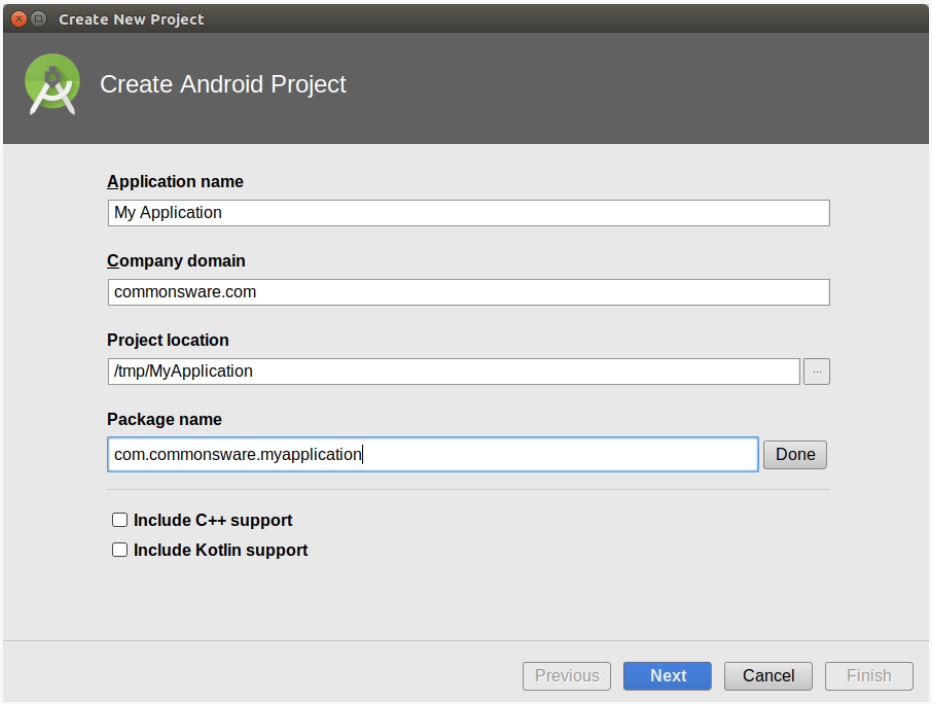


Figure 13: Android Studio Create-Project Wizard, First Page, with Editable Package Name

Clicking “Next” will advance you to a wizard page where you indicate what sort of project you are creating, in terms of intended device type (phones/tablets, TVs, etc.) and minimum required SDK level:

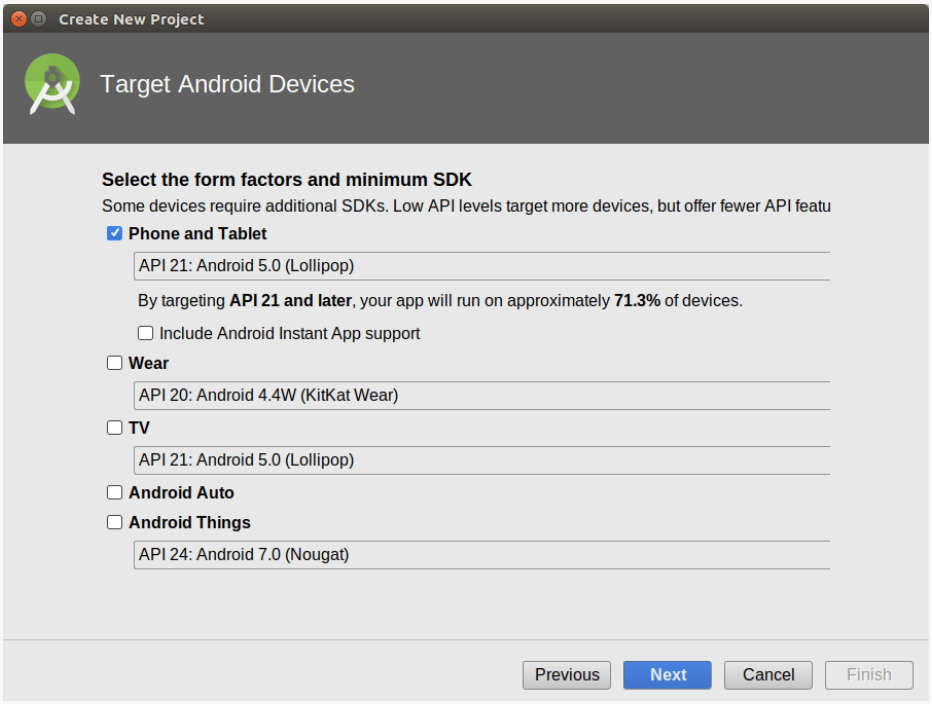


Figure 14: Android Studio Create-Project Wizard, Second Page

The "Minimum SDK" refers to how far back in Android's version history you are willing to support. The lower the value you specify here, the more Android devices can run your app, but the more work you will have to do to test whether your app really does support those devices.

Developers just starting out on Android should only check "Phone and Tablet" as the device type, leaving the other checkboxes unchecked. The default "Minimum SDK" value also usually is a good choice, and it can be changed readily in your project, as we will see later in the book.

Clicking "Next" advances you to the third page of the wizard, where you can choose if Android Studio should create an initial activity for you, and if so, based on what template:

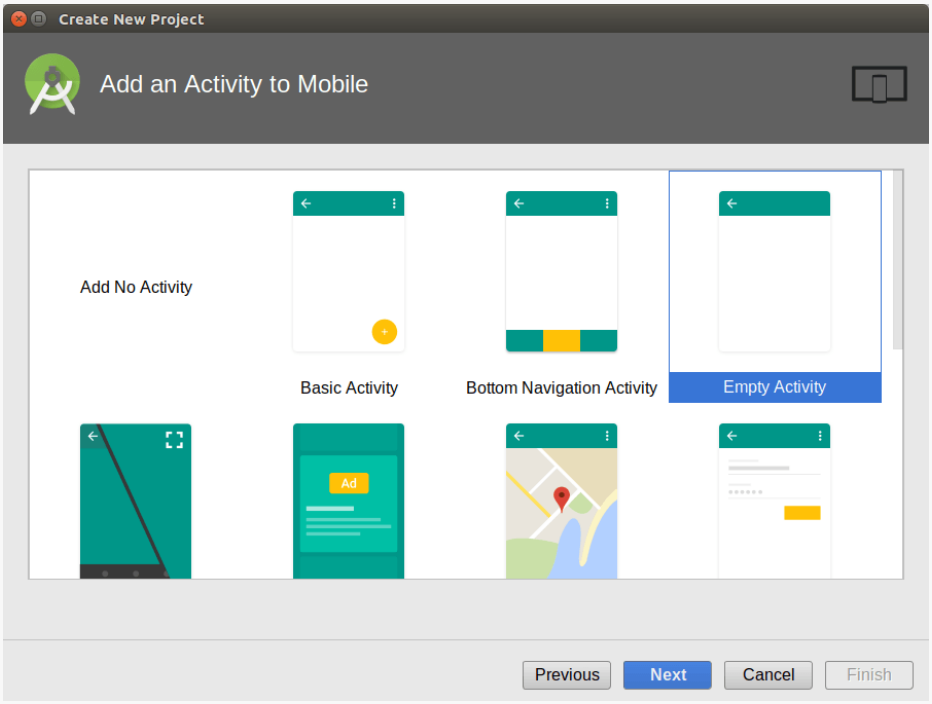


Figure 15: Android Studio Create-Project Wizard, Third Page

None of these templates are especially good, as they add a lot of example material that you will wind up replacing. "Empty Activity" is the best of the available options for first-time Android developers, simply because it adds the least amount of this "cruft".

If you choose any option other than "Add No Activity", clicking "Next" will advance you to a page in the wizard where you can provide additional details about the activity to be created:

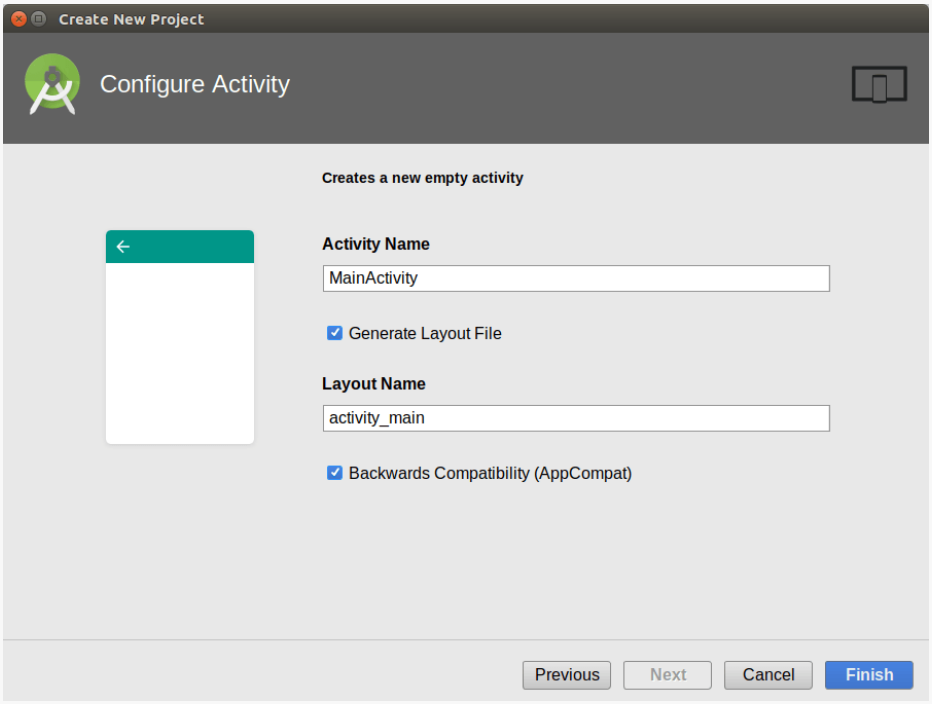


Figure 16: Android Studio Create-Project Wizard, Fourth Page

What options appear here will vary based upon the template you chose in the previous page. Common options include "Activity Name" (the name of the Java class for your activity) and "Layout Name" (the base name of an XML file that will contain a UI definition of your activity).

The "Backwards Compatibility (AppCompat)" checkbox indicates if you want to use a library known as AppCompat. We will discuss using libraries later in the book, as well as what this "AppCompat" is. Unless you know for certain that you want to use AppCompat — and few of this book's example apps do — uncheck this checkbox.

Clicking "Finish" will generate your project files.

Copying a Project

Android Studio projects are simply directories of files, with no special metadata held elsewhere. Hence, to copy a project, just copy its directory.

Importing a Project

You can import a project from one of two places:

- If you are at the initial dialog that you first encountered when you opened Android Studio, choose the "Import Project..." menu item
- If you are inside the Android Studio IDE itself, choose File > New... > Import Project... from the main menu

Then, choose the directory containing the project to be imported.

What happens now depends upon the nature of the project. If the project was already set up for use with Android Studio, or at least with the Android Gradle Plugin, the Android Studio-specific files will be created (or updated) in the project directory.

However, if the project was not set up for Android Studio (or at least for the Android Gradle Plugin), but does have Eclipse project files (or at least a `project.properties` file), you will be led through an Eclipse import wizard. This will be fairly uncommon nowadays.

Starter Project Generators

In addition to creating projects through an IDE's new-project wizard, there are various Web sites that offer online project generators:

- Android Bootstrap
- Android Kickstartr

On those sites, you provide basic configuration data, such as your application's package name, and they generate a complete starter project for you. These projects tend to be significantly more advanced than what you get from the IDE wizards. On the plus side, you get a more elaborate "scaffold" on which you can "hang" your own business logic. However, understanding what those generators create and how to change the generated code requires a fair bit of Android development experience.

Tutorial #2 - Creating a Stub Project

Creating an Android application first involves creating an Android "project". As with many other development environments, the project is where your source code and other assets (e.g., icons) reside. And, the project contains the instructions for your tools for how to convert that source code and other assets into an Android APK file for use with an emulator or device, where the APK is Android's executable file format.

Hence, in this tutorial, we kick off development of a sample Android application, to give you the opportunity to put some of what you are learning in this book in practice.

About Our Tutorial Project

The application we will be building in these tutorials is called `EmPubLite`. `EmPubLite` will be a digital book reader, allowing users to read a digital book like the one that you are reading right now.

`EmPubLite` will be a partial implementation of the `EmPub` reader used for the APK version of this book. `EmPub` itself is a fairly extensive application, so `EmPubLite` will have only a subset of its features.

The "Em" of `EmPub` and `EmPubLite` stands for "embedded". These readers are not designed to read an arbitrary EPUB or MOBI formatted book that you might download from somewhere. Rather, the contents of the book (largely an unpacked EPUB file) will be "baked into" the reader APK itself, so by distributing the APK, you are distributing the book.

About the Rest of the Tutorials

Of course, you may have little interest in writing a digital book reader app.

The tutorials presented in this book are certainly optional. There is no expectation that you have to write any code in order to get value from the book. These tutorials are here simply as a way to help those of you who "learn by doing" have an opportunity to do just that.

Hence, there are any number of ways that you can use these tutorials:

- You can ignore them entirely. That is not the best answer, but you are welcome to do it.
- You can read the tutorials but not actually do any of the work. This is the best low-effort answer, as it is likely that you will learn things from the tutorials that you might have missed by simply reading the non-tutorial chapters.
- You can follow along the steps and actually build the `EmPubLite` app.
- You can download the answers from the book's GitHub repository. There, you will find one directory per tutorial, showing the results of having done the steps in that tutorial. For example, you will find a `T2-Project/` directory containing a copy of the `EmPubLite` sample app after having completed the steps found in this tutorial. You can import these projects into your IDE, examine what they contain, cross-reference them back to the tutorials themselves, and run them.

Any of these are valid options — you will need to choose for yourself what you wish to do.

About Our Tools

The instructions in the remaining tutorials should be accurate for Android Studio 3.0.x. The instructions *may* work for other versions of this IDE, but there may also be some differences.

Step #1: Importing the Project

We need to create the Android project for `EmPubLite`.

Normally, you would use the new-project wizard to create a new project. However, the problem with the new-project wizard is that Google keeps changing what the new-project wizard generates. In most situations, that is not a huge problem. However, it becomes a problem for tutorials like this one, as if Google changes what is in the new project, the tutorial's instructions become out of date.

So, instead, we will import an existing project, so we can start from a stable base.

Visit the releases page of this book's GitHub repository. Then, scroll down to this book's version and download the `EmPubLite-Starter.zip` file for it. UnZIP that project to some place on your development machine. It will unZIP into an `EmPubLite/` directory.

Then, import the project. From the Android Studio welcome dialog, that is handled by the "Import project (Eclipse ADT, Gradle, etc.)" option. From an existing open Android Studio IDE window, you would use File > New > Import Project... from the main menu.

Importing a project brings up a typical directory-picker dialog. Pick the `EmPubLite/` directory and click OK to begin the import process. This may take a while, depending on the speed of your development machine. A "Tip of the Day" dialog may appear, which you can dismiss.

At this point, you should have an empty Android Studio IDE window, probably with an error message:

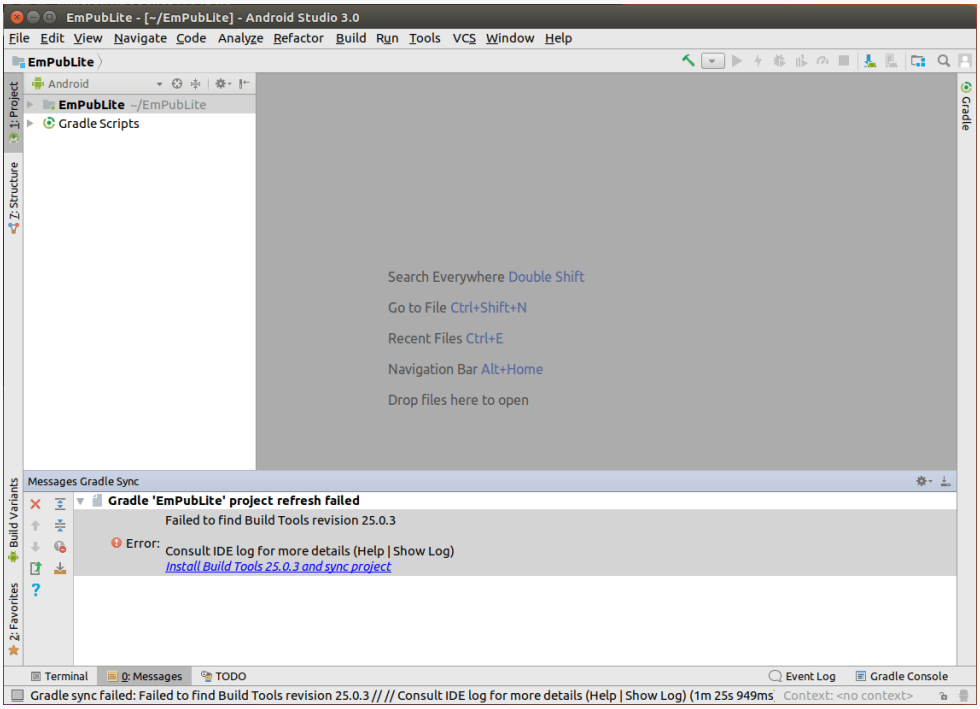


Figure 17: Android Studio, As Initially Launched, Showing Error

In the "Messages Gradle Sync" tool pane, towards the bottom of the screen, you may have a "Failed to find Build Tools revision 25.0.3". Each Android project states what version of the development tools it uses, and this one uses a version slightly older than the current. In that tool pane, the "Install Build Tools 25.0.3 and sync project" message is a hyperlink — click that to download the missing bits. This is a common pattern with these error messages, where the message will contain a link to try to fix the problem.

Next, after several moments, you will get a pop-up dialog like this one:

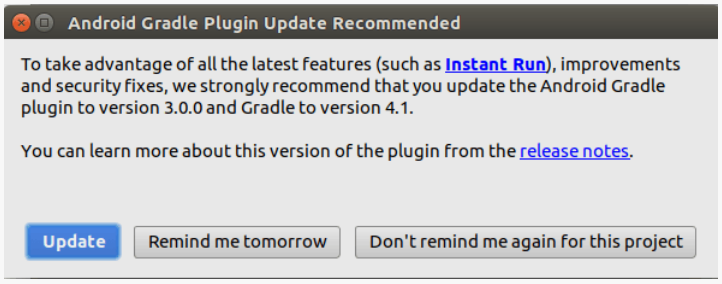


Figure 18: Android Studio, Update Recommendation Dialog

The trigger for this dialog is similar to the trigger for the preceding error message: the project that you imported is looking to use some older tools than what is the latest-and-greatest. For projects that you create yourself from scratch, you will not get this message, as the new-project wizard always uses the latest tools... even if that is not always the best idea. For projects like this one, that you import from elsewhere, this dialog is common. For the purposes of these tutorials, click the "Don't remind me again for this project" button. For other projects that you import, you could opt to click the "Update" button, which will adjust the project to use the newer tools, though this may cause compatibility problems that you would have to debug.

The "Project" tool — docked by default on the left side, towards the top — brings up a way for you to view what is in the project. Android Studio has several ways of viewing the contents of Android projects. The default one, that you are presented with when creating or importing the project, is known as the "Android view":

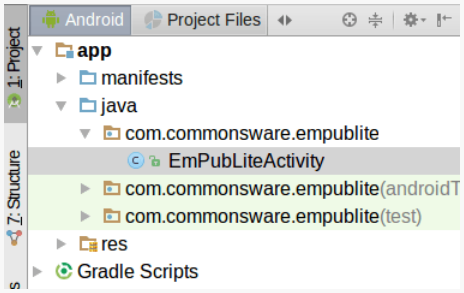


Figure 19: Android Studio "Android View"

While you are welcome to navigate your project using it, the tutorial chapters in this book, where they have screenshots of Android Studio, will show the project view:

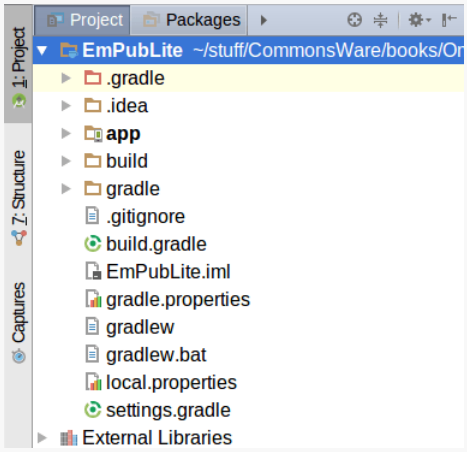


Figure 20: Android Studio "Project View"

To switch to this view — and therefore match what the tutorials will show you — click the "Android" entry above the tree, to display a drop-down of different views. Choose "Project", and that will switch you to the project view used by these tutorials.

Step #2: Get Ready for the x86 Emulator

Your first decision to make is whether or not you want to bother setting up an emulator image right now. If you have an Android device, you may prefer to start testing your app on it, and come back to set up the emulator at a later point. In that case, skip to Step #4.

Otherwise, here is what you may need to do, based on the operating system on your development machine.

Windows

If your CPU met the requirements, and you successfully enabled the right things in your system's BIOS, the Android Studio installation should have installed HAXM, and you should be ready to go.

If, on the other hand, you got some error messages in the installation wizard regarding HAXM, you would need to address those first.

Mac

The wizards of Cupertino set up their Mac hardware to be able to run the Android x86 emulator, which is awfully nice of them, considering that Android competes with iOS. The Android Studio installation wizard should have installed HAXM successfully, and you should be able to continue with the next step of the tutorial.

Linux

The Android x86 emulator on Linux does not use HAXM. Instead, it uses KVM, a common Linux virtualization engine.

If, during the Android Studio installation process, the wizard showed you a page that said that you needed to configure KVM, you will need to do just that before you can set up and use the x86 emulator. The details of how to set up KVM will vary by Linux distro (e.g., Ubuntu).

Step #3: Set Up the AVD

The Android emulator can emulate one or several Android devices. Each configuration you want is stored in an "Android virtual device", or AVD. The AVD Manager is where you create these AVDs.

Note that Android Studio now has its own implementation of the AVD Manager that is separate from the one Android developers have traditionally used. You may see screenshots of the older AVD Manager in blog posts, Stack Overflow answers, and the like. The AVD Manager still fills the same role, but it has a different look and feel.

To open the AVD Manager in Android Studio, choose Tools > Android > AVD Manager from the main menu.

You should be taken to "welcome"-type screen:

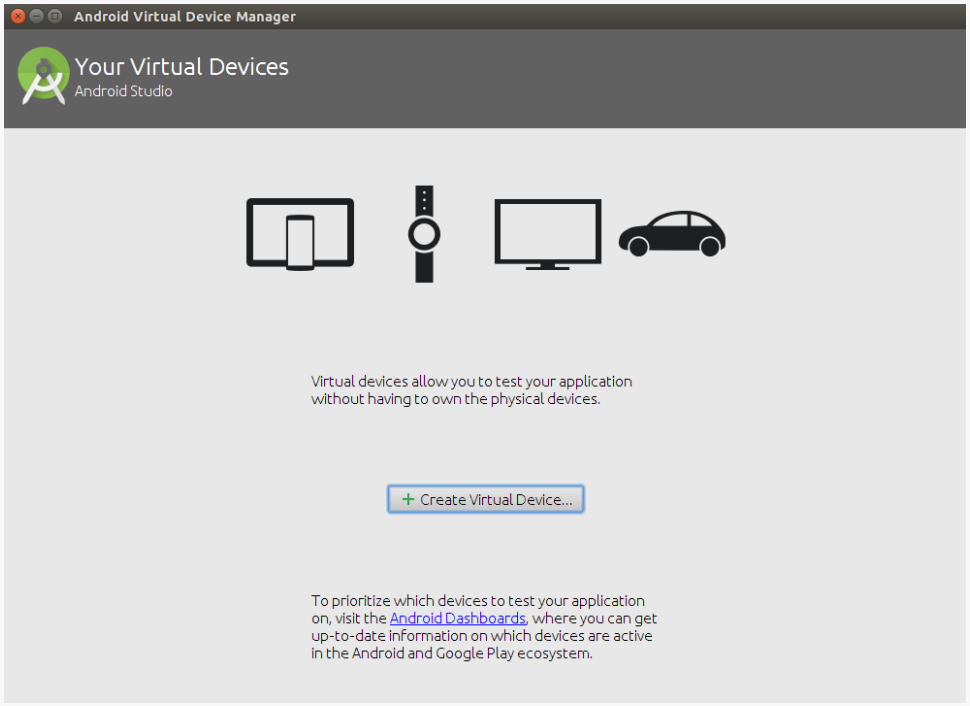


Figure 21: Android Studio AVD Manager, Welcome Screen

Click the "Create Virtual Device" button, which brings up a "Virtual Device Configuration" wizard:

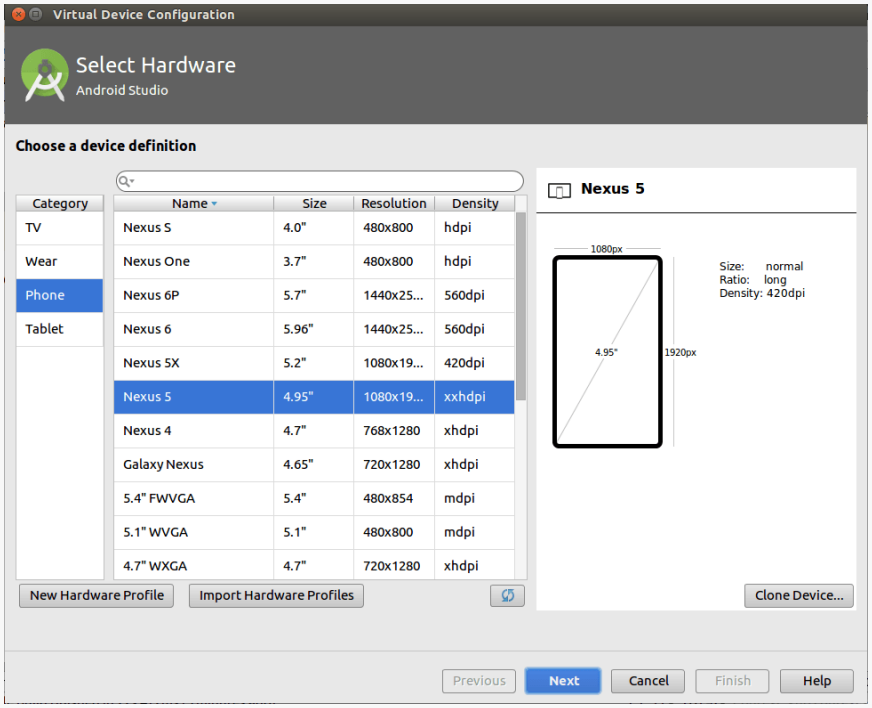


Figure 22: Android Studio Virtual Device Configuration Wizard, First Page

The first page of the wizard allows you to choose a device profile to use as a starting point for your AVD. The "New Hardware Profile" button allows you to define new profiles, if there is no existing profile that meets your needs.

Since emulator speeds are tied somewhat to the resolution of their (virtual) screens, you generally aim for a device profile that is on the low end but is not completely ridiculous. For example, an 800x480 or 1280x768 phone would be considered by many people to be fairly low-resolution. However, there are plenty of devices out there at that resolution (or lower), and it makes for a reasonable starting emulator.

If you want to create a new device profile based on an existing one — to change a few parameters but otherwise use what the original profile had – click the "Clone Device" button once you have selected your starter profile.

However, in general, at the outset, using an existing profile is perfectly fine. The Galaxy Nexus or Nexus 4 images are likely choices to start with.

Clicking "Next" allows you to choose an emulator image to use:

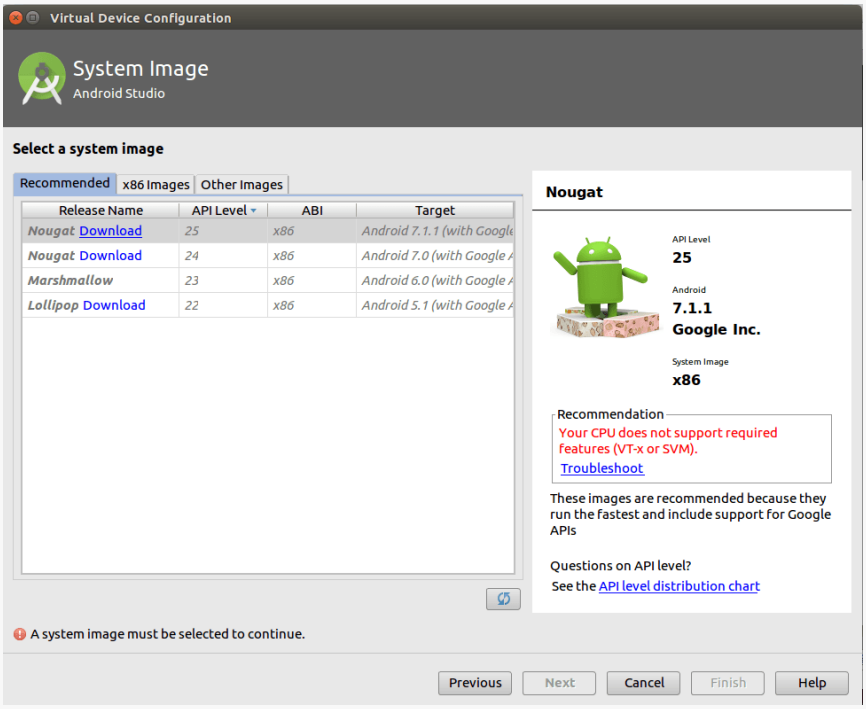


Figure 23: Android Studio Virtual Device Configuration Wizard, Second Page

The emulator images are spread across three tabs:

- "Recommended"
- "x86 Images"
- "Other Images"

For the purposes of the tutorials, you do not need an emulator image with the "Google APIs" — those are for emulators that have Google Play Services in them and related apps like Google Maps. However, in terms of API level, you can choose anything from API Level 15 (Android 4.0.3) on up. You should have one or more suitable images already set up for you, courtesy of having installed Android Studio.

If you click on the x86 Images tab, you should see some images with a "Download" link, and others without it:

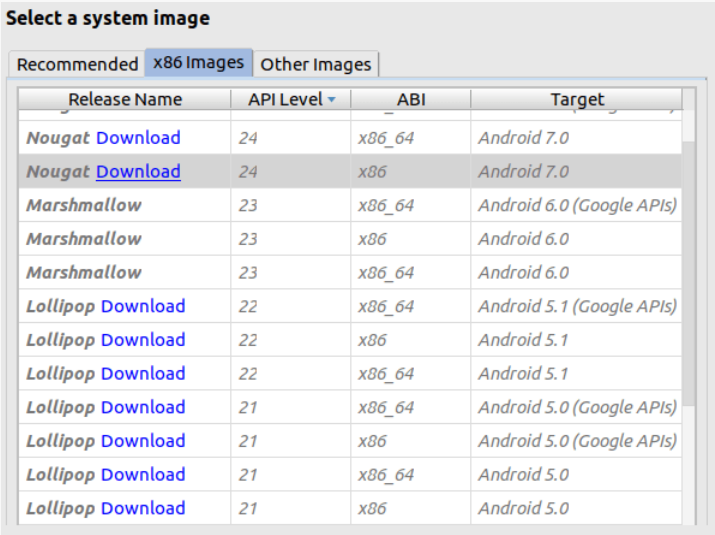


Figure 24: Android Studio Virtual Device Configuration Wizard, x86 Images

The emulator images with "Download" next to them will trigger a one-time download of the files necessary to create AVDs for that particular API level and CPU architecture combination, after another license dialog and progress dialog:

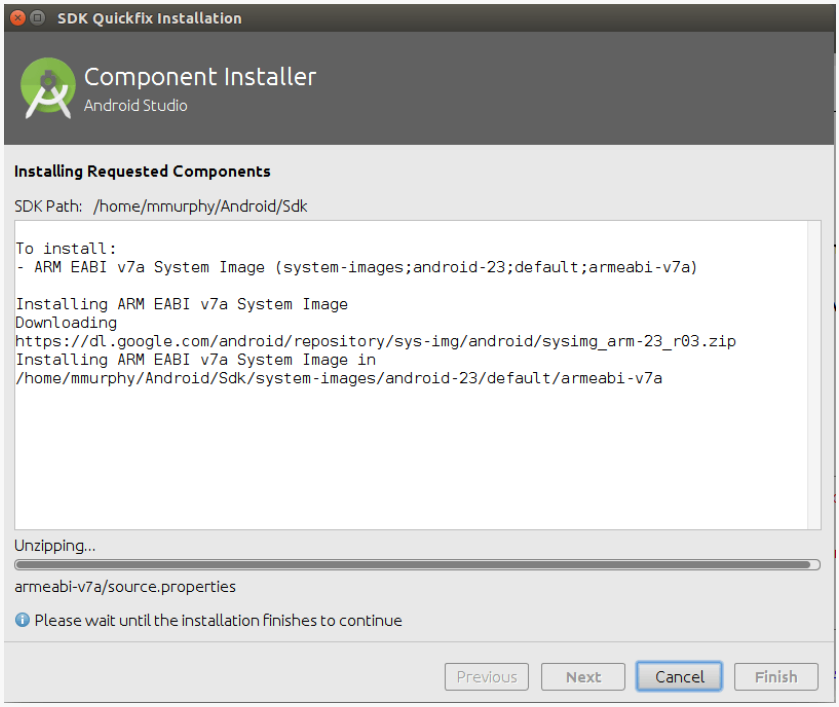


Figure 25: Android Studio Component Installer Dialog, Downloading API 23 ARM Image

Once you have identified the image(s) that you want — and have downloaded any that you did not already have — click on one of them in the wizard:

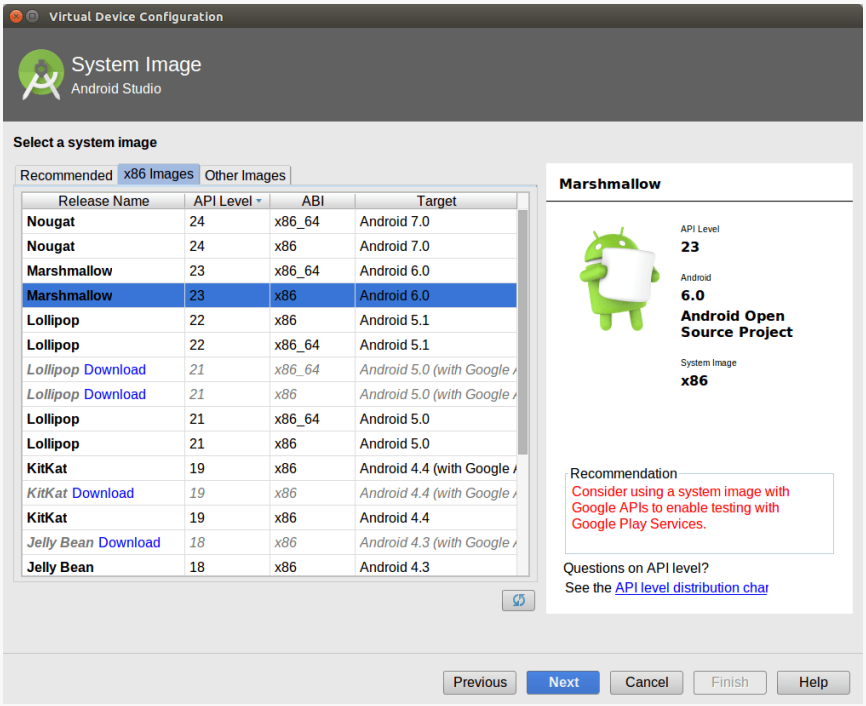


Figure 26: Android Studio Virtual Device Configuration Wizard, After Choosing Image

Clicking "Next" allows you to finalize the configuration of your AVD:

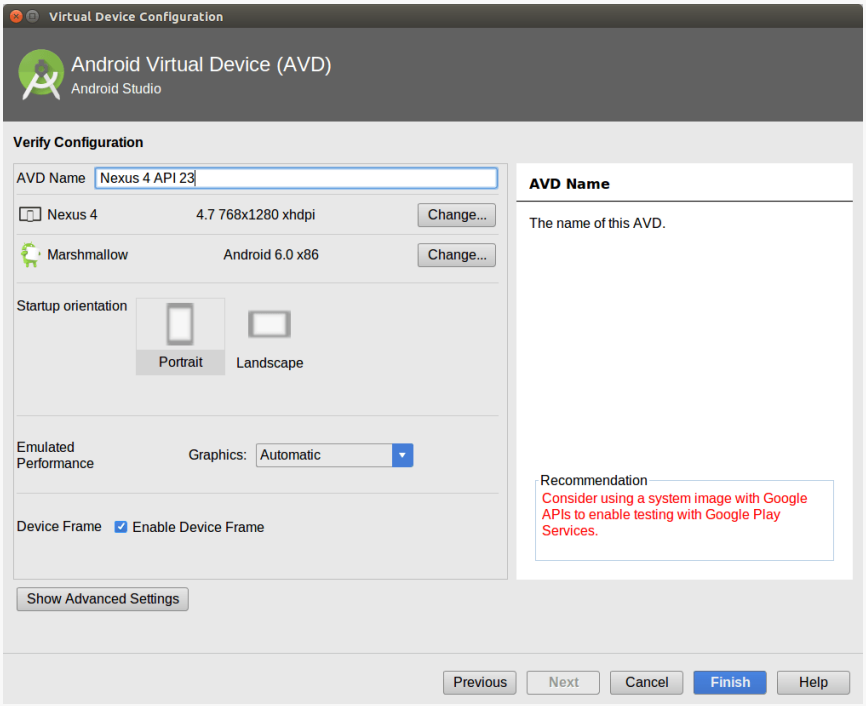


Figure 27: Android Studio Virtual Device Configuration Wizard, Third Page

A default name for the AVD is suggested, though you are welcome to replace this with your own value.

Change the AVD name, if necessary, to something valid: only letters, numbers, spaces, and select punctuation (e.g., `.`, `_`, `-`, `(`, `)`) are supported.

The rest of the default values should be fine for now.

Clicking "Finish" will return you to the main AVD Manager, showing your new AVD. You can then close the AVD Manager window.

Step #4: Set Up the Device

You do not need an Android device to get started in Android application development. Having one is a good idea before you try to ship an application (e.g., upload it to the Play Store). And, perhaps you already have a device – maybe that is what is spurring your interest in developing for Android.

If you do not have an Android device that you wish to set up for development, skip this step.

The first step to make your device ready for use with development is to go into the Settings application on the device. What happens now depends a bit on your Android version:

- On Android 1.x/2.x, go into Applications, then into Development
- On Android 3.0 through 4.1, go into "Developer options" from the main Settings screen
- On Android 4.2 and higher, go into About, tap on the build number seven times, then press BACK, and go into "Developer options" (which was formerly hidden)

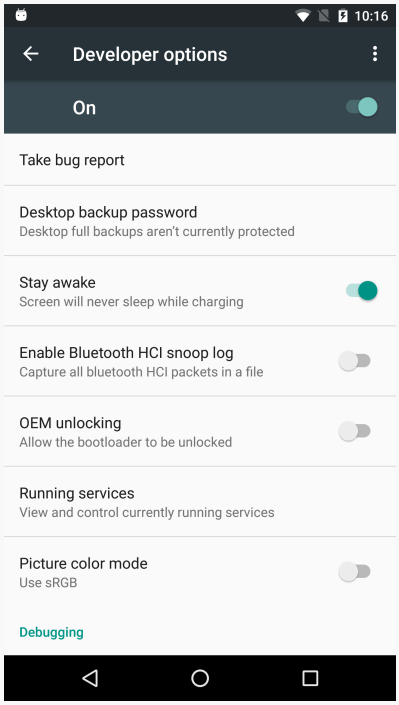


Figure 28: Developer Options, in Settings App

You may need to slide a switch in the upper-right corner of the screen to the "ON" position to modify the values on this screen.

Generally, you will want to scroll down and enable USB debugging, so you can use your device with the Android build tools:

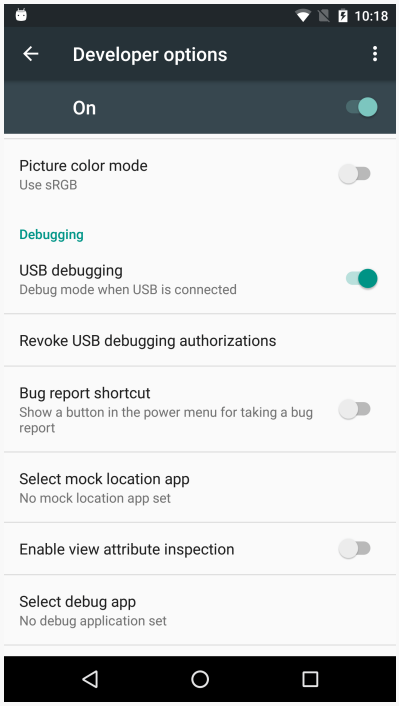


Figure 29: Debugging Options, in Settings App

You can leave the other settings alone for now if you wish, though you may find the "Stay awake" option to be handy, as it saves you from having to unlock your phone all of the time while it is plugged into USB.

Note that on Android 4.2.2 and higher devices, before you can actually use the setting you just toggled, you will be prompted to allow USB debugging with your *specific* development machine via a dialog box:

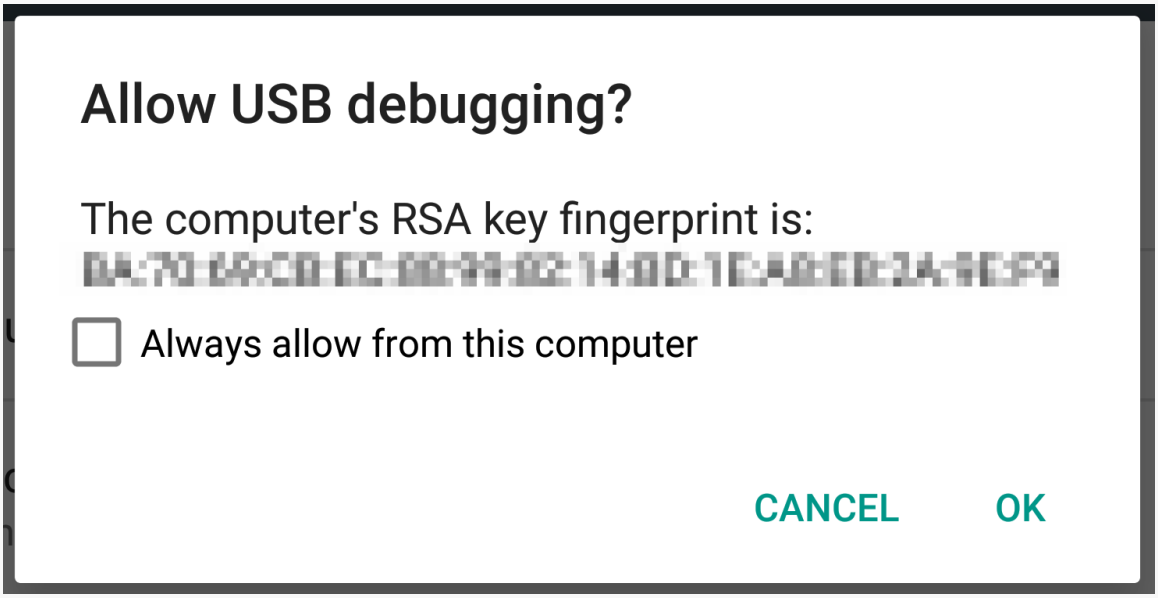


Figure 30: Allow USB Debugging Dialog

This occurs when you plug in the device via the USB cable and have the driver appropriately set up. That process varies by the operating system of your development machine, as is covered in the following sections.

Windows

When you first plug in your Android device, Windows will attempt to find a driver for it. It is possible that, by virtue of other software you have installed, that the driver is ready for use. If it finds a driver, you are probably ready to go.

If the driver is not found, here are some options for getting one.

Windows Update

Some versions of Windows (e.g., Vista) will prompt you to search Windows Update for drivers. This is certainly worth a shot, though not every device will have supplied its driver to Microsoft.

Standard Android Driver

In your Android SDK installation, if you chose to install the "Google USB Driver" package from the SDK Manager, you will find an `extras/google/usb_driver/`

directory, containing a generic Windows driver for Android devices. You can try pointing the driver wizard at this directory to see if it thinks this driver is suitable for your device. This will often work for Nexus devices.

Manufacturer-Supplied Driver

If you still do not have a driver, the OEM USB Drivers in the developer documentation may help you find one for download from your device manufacturer. Note that you may need the model number for your device, instead of the model name used for marketing purposes (e.g., GT-P3113 instead of "Samsung Galaxy Tab 2 7.0").

macOS and Linux

Odds are decent that simply plugging in your device will "just work". You can see if Android recognizes your device via running `adb devices` in a shell (e.g., macOS Terminal), where `adb` is in your `platform-tools/` directory of your SDK. If you get output similar to the following, the build tools detected your device:

```
List of devices attached
HT9CPP809576  device
```

If you are running Ubuntu (or perhaps other Linux variants), and this command did not work, you may need to add some `udev` rules. For example, here is a `51-android.rules` file that will handle the devices from a handful of manufacturers:

```
SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="22b8", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="18d1", MODE="0666"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="18d1", ATTRS{idProduct}=="0c01", MODE="0666",
OWNER="[me]"
SUBSYSTEM=="usb", SYSFS{idVendor}=="19d2", SYSFS{idProduct}=="1354", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="04e8", SYSFS{idProduct}=="681c", MODE="0666"
```

Drop that in your `/etc/udev/rules.d` directory on Ubuntu, then either reboot the computer or otherwise reload the `udev` rules (e.g., `sudo service udev reload`). Then, unplug and re-plug in the device and see if it is detected.

Step #5: Running the Project

Now, we can confirm that our project is set up properly by running it on a device or emulator.

To do that in Android Studio, just press the Run toolbar button (usually depicted as a green rightward-pointing triangle).

You will then be presented with a dialog indicating where you want the app to run: on some existing device or emulator, or on some newly-launched emulator:

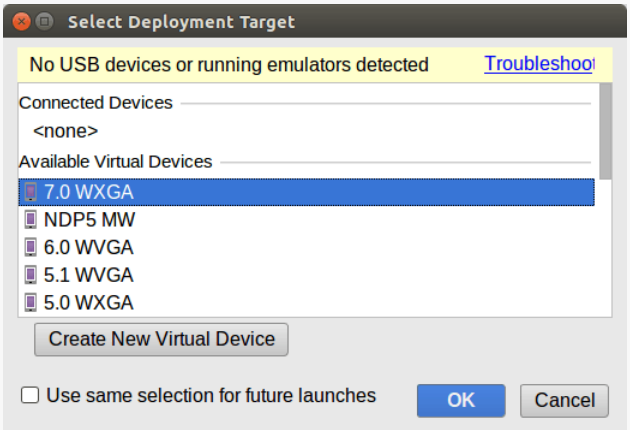


Figure 31: Android Studio Device Chooser Dialog

If you do not have an emulator running, choose one from the list, then click OK. Android Studio will launch your emulator for you.

And, whether you start a new emulator instance or reuse an existing one, your app should appear on it:

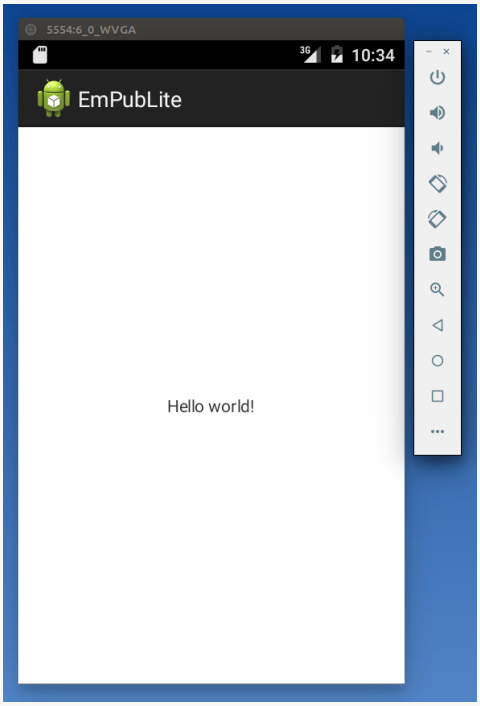


Figure 32: Android 7.0 Emulator with EmPubLite

Note that you may have to unlock your device or emulator to actually see the app running.

The first time you launch the emulator for a particular AVD, you may see this message:

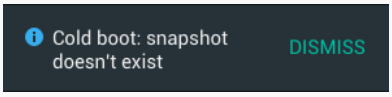


Figure 33: Android Emulator Cold-Boot Warning

The emulator now behaves a bit more like an Android device. Closing the emulator window used to be like completely powering off a phone, but now it is more like tapping the POWER button to turn off the screen. The next time you start that particular AVD, it will wake up to the state in which you left it, rather than booting from scratch ("cold boot"). This speeds up starting the emulator. Occasionally, though, you will have the need to start the emulator as if the device were powering on. To do that, in the AVD Manager, in the drop-down menu in the Actions column, choose "Cold Boot Now".

In Our Next Episode…

... we will modify the `AndroidManifest.xml` file of our tutorial project.

Getting Around Android Studio

This chapter will serve as a quick tour of the Android Studio IDE, to help you get settled in. Other Android-specific capabilities of Android Studio will be explored in the chapters that follow.

Navigating The Project Explorer

After the main editing area — where you will modify your Java source code, your resources, and so forth — the piece of Android Studio you will spend the most time with is the project explorer, usually available on the left side of the IDE window:

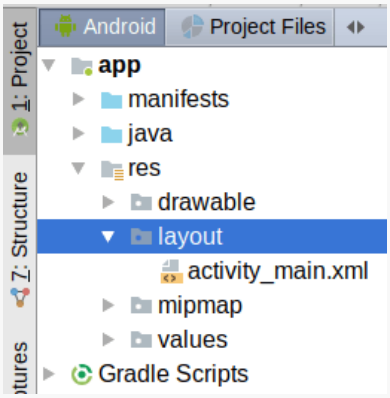


Figure 34: Android Studio Project Explorer, Showing Android Project View

This explorer pane has two main "project views" that an Android developer will use: the Android project view and the classic project view.

Android Project View

By default, when you create or import a project, you will wind up in the Android project view.

In theory, the Android project view is designed to simplify working with Android project files. In practice, it may do so, but only for some advanced developers. On the whole, it makes the IDE significantly more complicated for newcomers to Android, as it is rather difficult to see where things are and what relates to what.

We will return to the Android project view a bit later in the book and explain its benefits relative to resources and Gradle's source sets.

However, for most of the book — most importantly, for the tutorials – we will use the classic project view.

Classic Project View

To switch to the classic project view, click the pair of arrowheads to the right of the "Project Files" tab just above the tree in the explorer, and choose Project:

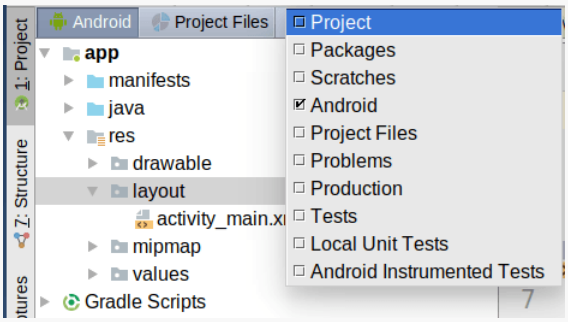


Figure 35: Android Studio Project Explorer, Showing Project View Drop-Down

On some machines, in some cases, the pair of arrowheads is instead a single drop-down arrowhead next to the currently-chosen view (e.g., Android).

Switching to the project view will change the contents of the tree to show you all of the files, in their associated directories:

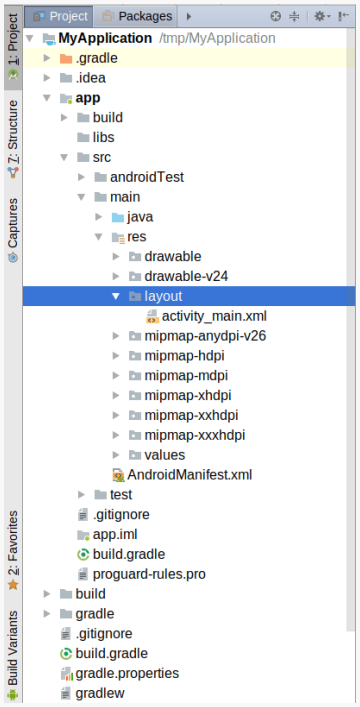


Figure 36: Android Studio Project Explorer, Showing Classic Project View

This project view is much like its equivalent in other IDEs, allowing you to find all of the pieces of your Android project. We will be exploring what those pieces are, and how their files are organized in our projects, in the next chapter.

Context Menus in the Explorer

Right-clicking over a directory or file in the explorer will give you a context menu with a variety of options. Some of these will be typical of any sort of file manager, such as “cut”, “copy”, and/or “paste” options. Some of these will be organized according to how Android Studio manages application development, such as the “Refactor” sub-menu, where you can rename or move files around. Yet others will be specific to Android Studio, such as the ability to invoke wizards to create certain types of Android components or other Java classes.

Opening Files from the Explorer

Double-clicking on a file usually opens that file in a tab that allows you to edit that file, using some sort of editor.

Some file types, like images, can be opened but not edited, as Android Studio does not have editors for all file types.

Running Projects

Of course, as you change your app, you will want to try it out and see if it works, whether on a device or an emulator.

The Basics

As noted in Tutorial #2, to run your project, just press the Run toolbar button:

Figure 37: Android Studio Run Controls, Showing Green Arrow to Run the App

You will then be presented with a dialog indicating where you want the app to run:

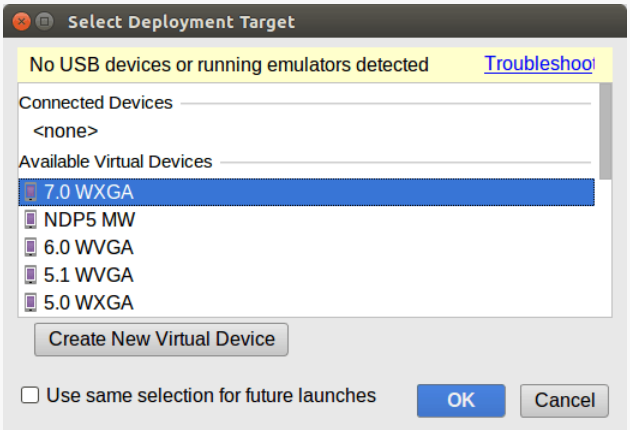


Figure 38: Android Studio Device Chooser Dialog

The "Connected Devices" category lists any devices or running emulators that the build tools can find. Some may be disabled due to compatibility issues, such as having an emulator for an old version of Android where your app requires a newer version of Android.

The "Available Emulators" category lists all AVDs that you have defined in the AVD Manager that are not already running. Again, you may find that some are disabled for compatibility reasons.

The “Create New Emulator” button brings up the wizard to create a new AVD, just like the one you can launch from the AVD Manager.

“Instant Run”

Next to the green “run” arrow button in the toolbar is a lightning bolt button. Sometimes, this will be grayed out and unusable. Other times, it will appear in yellow:

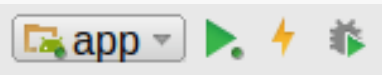


Figure 39: Android Studio Run Controls, With Instant Run Enabled

This button performs what is called “Instant Run”. Instead of building your app and pushing the app to the device or emulator, Instant Run attempts to patch your existing app based on whatever changes you made to the project since you last ran it.

On the plus side, Instant Run is very fast. However, the patched app is not exactly the same as would be your app built from scratch. Feel free to use this for smaller changes if you wish.

Viewing Output

Beyond your app itself, Android Studio will generate other sorts of diagnostic output, in the form of “console”-style transcripts of things that have occurred. The two of these that probably will matter most for you are the Gradle console and Logcat.

Gradle Console

By default, docked in the lower-right corner of your Android Studio window is a “Gradle Console” item. Tapping on that will open up a pane showing the output of attempts to build your application:

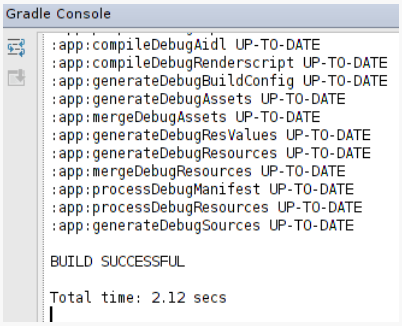


Figure 40: Android Studio Gradle Console

This may automatically appear from time to time, if specific build problems are detected, and you can always go examine it whenever you need.

Click on the "Gradle Console" item again to collapse the view and get it out of your way.

Logcat

Messages that appear at runtime — including the all-important Java stack traces triggered by bugs in your code — are visible in Logcat. The Logcat tool docked towards the lower-left corner of your Android Studio window will display Logcat when tapped:

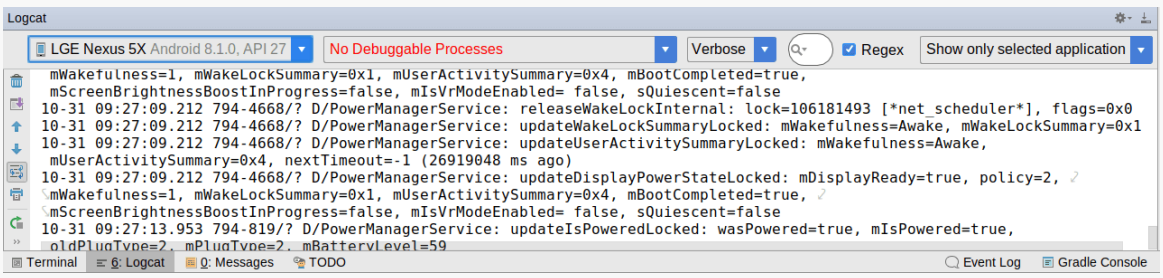


Figure 41: Android Studio Logcat Tool

Logcat is explained in greater detail a bit later in this book.

Accessing Android Tools

Not everything related to Android is directly part of Android Studio itself. In some cases, tools need to be shared between users of Android Studio, users of Eclipse, and

users of "none of the above". In some cases, while the long term direction may be to incorporate the tools' functionality directly into Android Studio, that work simply has not been completed to date.

Here are some noteworthy Android-related tools that you can access via the Tools > Android main menu option.

SDK and AVD Managers

As we saw in Tutorial #1, the SDK Manager is Android's tool for downloading pieces of the Android SDK, including:

- "SDK Platform" editions, allowing us to compile against a particular API level
- ARM and (sometimes) x86 emulator images
- Documentation
- Updates to the core build tools
- Etc.

You can launch the SDK Manager via Tools > Android > SDK Manager from the Android Studio main menu, or by clicking on the "droid in a box" toolbar button:

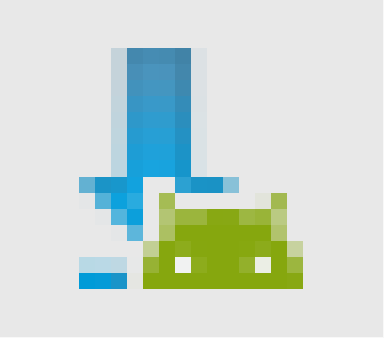


Figure 42: Android Studio SDK Manager Toolbar Icon

The AVD Manager is the tool for creating emulators that emulate certain Android environments, based upon API level, screen size, and other characteristics.

You can launch the AVD Manager via Tools > Android > AVD Manager from the Android Studio main menu, or by clicking the "droid and a screen" toolbar button:

Figure 43: Android Studio AVD Manager Toolbar Icon

Android Studio and Release Channels

When you install Android Studio for the first time, your installation will be set up to get updates on the "stable" release channel. Here, a "release channel" is a specific set of possible upgrades. The "stable" release channel means that you are getting full production-ready updates. Android Studio will check for updates when launched, and you can manually check for updates via the main menu (e.g., Help > Check for Update... on Windows and Linux).

If an update is available, you will be presented with a dialog box showing you details of the update:

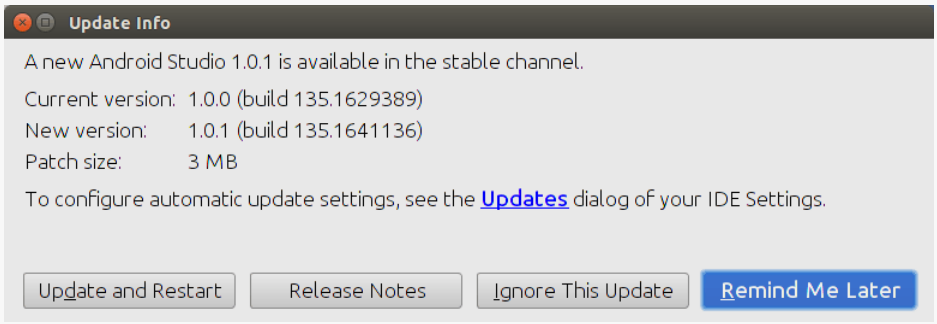


Figure 44: Android Studio Update Dialog

Choosing "Release Notes" will bring up a Web page with release notes for the new release. Clicking "Update and Restart" does pretty much what the button name suggests: it downloads the update and restarts the IDE, applying the update along the way.

Clicking the "Updates" hyperlink in the dialog brings up yet another dialog, allowing you to choose which release channel you want to subscribe to:

Figure 45: Android Studio Update Release Channel Dialog

You have four channels to choose from:

- Stable, which is appropriate for most developers
- Beta, which will get updates that are slightly ahead of stable
- Dev, which is even more ahead than is the beta channel
- Canary, which is updated *very* early (and the name, suggestive of a "canary in a coal mine", indicates that you are here to help debug the IDE)

Visit the Trails!

Android Studio's Project Structure dialog and Translations Editor are covered later in this book.

Contents of Android Projects

The Android build system is organized around a specific directory tree structure for your Android project, much like any other Java project. The specifics, though, are fairly unique to Android — the Android build tools do a few extra things to prepare the actual application that will run on the device or emulator.

Here is a quick primer on the project structure, to help you make sense of it all, particularly for the sample code referenced in this book.

What You Get, In General

The details of *exactly* what files are in your project depend on a variety of things:

- What IDE and version the project was used to create the project
- What project template was used to create the project
- What other changes have been made to the project after it was created

However, usually, there are many elements in common.

The Modules

A default Android Studio project will have an `app/` directory off of the project root directory (i.e., the directory in which you told Android Studio to create the project). That `app/` directory represents what Android Studio refers to as a module. Most projects just have this one `app/` module, though you can have more than one module if needed.

The Source Sets

Inside of `app/` you will find a `src/main/` directory. This represents the `main` source set, and it contains all of the "source" that your project is contributing to build the application. Here, "source" means more than just programming language source code (e.g., Java). It also includes other types of files, such as resources and the manifest, that contribute to the app.

The Manifest

Inside of `app/src/main/`, you will find a file named `AndroidManifest.xml`. `AndroidManifest.xml` is an XML file describing the application being built and what components — activities, services, etc. — are being supplied by that application. You can think of it as being the "table of contents" of what your application is about, much as a book has a "table of contents" listing the various parts, chapters, and appendices that appear in the book.

We will examine the manifest a bit more closely starting in the next chapter.

The Java

When you created the project, if you had Android Studio create an activity for you, you supplied the class name of the "main" activity for the application (e.g., `MainActivity`). That will be combined with the package name that you provided for the project to determine the fully-qualified class name for this activity (e.g., `com.commonsware.android.MainActivity`).

You will find that your project's Java source tree already has the package's directory tree in place, plus a class representing your requested activity (e.g., `com/commonsware/android/MainActivity.java`). This will be inside the `main` source set, so the full path from the project root would be something like `app/src/main/java/com/commonsware/android/MainActivity.java`.

You are welcome to modify this file and add Java classes as needed to implement your application, and we will demonstrate that countless times as we progress through this book.

Elsewhere — in directories that you normally do not work with — the Android build tools will also be code-generating some source code for you each time you build your app. One of the code-generated Java classes (`R.java`) will be important for

controlling our user interfaces from our own Java code, and we will see many references to this `R` class as we start building applications in earnest.

The Resources

You will also find that your project has a `res/` directory tree inside of the `main` source set. This holds "resources" — static files that are packaged along with your application, either in their original form or, occasionally, in a preprocessed form. Some of the subdirectories you will find or create under `res/` include:

1. `res/drawable/` and `res/mipmap-*/` (for a few values of `*`) for images (PNG, JPEG, etc.)
2. `res/layout/` for XML-based UI layout specifications
3. `res/menu/` for XML-based menu specifications
4. `res/raw/` for general-purpose files (e.g., an audio clip, a CSV file of account information)
5. `res/values/` for strings, dimensions, and the like
6. `res/xml/` for other general-purpose XML files you wish to ship

Some of the directory names may have suffixes, like `res/mipmap-hdpi/`. This indicates that the directory of resources should only be used in certain circumstances — in this case, the mipmap resources are designed for devices with high-density screens.

We will cover all of these, and more, later in this book.

The Build Instructions

The IDE needs to know how to take all of this stuff and come up with an Android APK file. Some of this is already "known" to the IDE based upon how the IDE was written. But some details are things that you may need to configure from time to time, and so those details are stored in files that you will edit, by one means or another, from your IDE.

In Android Studio, most of this knowledge is kept in files named `build.gradle`. These are for a build engine known as Gradle, that Android Studio uses to build APKs and other Android outputs.

More About the Directory Structure

All of those items are stored in a particular directory structure in an Android Studio project... at least by default. Android Studio and Gradle are powerful and can be configured to handle other structures, though most projects stick with the standard setup.

The Root Directory

In the root directory of your project, the most important item is the `app/` directory representing your application module. We will look at that more in the next section.

Beyond the `app/` directory, the other noteworthy files in the root of your project include:

- `build.gradle`, which is part of the build instructions for your project, as is described above
- Various other Gradle-related files (`settings.gradle`, `gradle.properties`, `gradlew`, `gradlew.bat`, and so forth)
- `local.properties`, which indicates where your Android SDK tools reside
- An `.iml` file, where Android Studio holds some additional metadata about your project
- A `.gitignore` file, representing standard rules for what should and should not go into version control, particularly with respect to the Git version control system

Eventually, you will have:

- A `build/` directory, containing the compiled output of your app, plus various reports and other files related to the build process and app testing
- A `.gradle/` directory, containing Gradle executable code
- An `.idea/` directory — this, along with the `.iml` file, represents data needed by IntelliJ IDEA, on which Android Studio is based

The App Directory

The `app/` directory, and its contents, are where you will spend most of your time as a developer. Only infrequently do you need to manipulate the files in the project root.

The most important thing in the `app/` directory is the `src/` directory, which is the root of your project's source sets. We will explore those more in the next section.

Beyond the `src/` directory, there are a few other items of note in `app/`:

- A `build/` directory, which will hold the outputs of building your app, including your APK file
- A `build.gradle` file, where most of your project-specific Gradle configuration will go, to teach Android Studio how to build your app
- An `app.iml` file, containing more Android Studio metadata
- Another `.gitignore` file, for overrides to the Git version control rules from the `.gitignore` file in the project root
- `proguard-rules.pro`, which are rules for a tool called ProGuard, which helps reduce the size of your app

The Source Sets

Source sets are where the "source" of your project is organized. As noted earlier, "source" not only refers to programming language source code (e.g., Java), but other types of inputs to the build, such as your resources.

The source set that you will spend most of your time in is `main/`. You may also have stub source sets named `androidTest` and `test`, for use in testing your app, as will be covered later in the book.

Inside of a source set, common items include:

- Java code, in a `java/` directory
- Resources, in a `res/` directory
- Assets, in an `assets/` directory, representing other static files you wish packaged with the application for deployment onto the device
- Your `AndroidManifest.xml` file

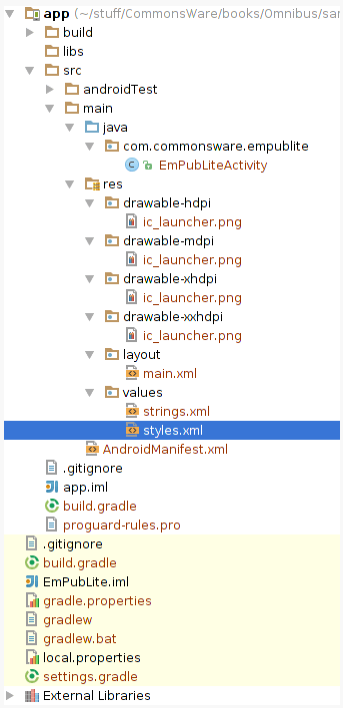


Figure 46: Android Studio Project Explorer, Showing EmPubLite

Some projects will have additional items, usually tied to specific add-on tools that the project happens to employ.

What You Get Out Of It

As part of running your app on a device or emulator, the IDE will generate an APK file. You will find this in the `build/outputs/apk` directory of your module's directory, (e.g., `app/build/outputs/apk` for a traditional Android Studio project).

The APK file is a ZIP archive containing your compiled Java classes along with packaged versions of the rest of your source (e.g., resources, manifest). The APK is what gets distributed to your users, whether through an app distribution service like Google's Play Store or through other means.

Introducing Gradle and the Manifest

In the discussion of Android Studio, this book has mentioned something called "Gradle", without a lot of explanation.

In this chapter, the mysteries of Gradle will be revealed to you.

(well, OK, *some* of the mysteries...)

We also mentioned in passing in the previous chapter the concept of the "manifest" — `AndroidManifest.xml` — as being a special file in our Android projects.

On the one hand, Gradle and the manifest are not strictly related. On the other hand, some (but far from all) of the things that we can set up in the manifest can be overridden in Gradle.

So, in this chapter, we will review both what Gradle is, what the manifest is, what each of their roles are, and the basics of how they tie together.

Gradle: The Big Questions

First, let us "set the stage" by examining what this is all about, through a series of fictionally-asked questions (FAQs).

What is Gradle?

Gradle is software for building software, otherwise known as "build automation software" or "build systems". You may have used other build systems before in other environments, such as `make` (C/C++), `rake` (Ruby), Ant (Java), Maven (Java), etc.

These tools know — via intrinsic capabilities and rules that you teach them — how to determine what needs to be created (e.g., based on file changes) and how to create them. A build system does not compile, link, package, etc. applications directly, but instead directs separate compilers, linkers, and packagers to do that work.

Gradle uses a domain-specific language (DSL) built on top of Groovy to accomplish these tasks.

What is Groovy?

There are many programming languages that are designed to run on top of the Java VM. Some of these, like JRuby and Jython, are implementations of other common programming languages (Ruby and Python, respectively). Other languages are unique, and Groovy is one of those.

Groovy scripts look a bit like a mashup of Java and Ruby. As with Java, Groovy supports:

- Defining classes with the `class` keyword
- Creating subclasses using `extends`
- Importing classes from external JARs using `import`
- Defining method bodies using braces (`{` and `}`)
- Objects are created via the `new` operator

As with Ruby, though:

- Statements can be part of a class, or simply written in an imperative style, like a scripting language
- Parameters and local variables are not typed
- Values can be automatically patched into strings, though using slightly different syntax (`"Hello, $name"` for Groovy instead of `"Hello, #{name}"` for Ruby)

Groovy is an interpreted language, like Ruby and unlike Java. Groovy scripts are run by executing a **`groovy`** command, passing it the script to run. The Groovy runtime, though, is a Java JAR and requires a JVM in order to operate.

One of Groovy's strengths is in creating a domain-specific language (or DSL). Gradle, for example, is a Groovy DSL for doing software builds. Gradle-specific capabilities appear to be first-class language constructs, generally indistinguishable

from capabilities intrinsic to Groovy. Yet, the Groovy DSL is largely declarative, like an XML file.

To some extent, we get the best of both worlds: XML-style definitions (generally with less punctuation), yet with the ability to "reach into Groovy" and do custom scripting as needed.

What Does Android Have To Do with Gradle?

Google has published the Android Gradle Plugin, which gives Gradle the ability to build Android projects. Google is also using Gradle and the Android Gradle Plugin as the build system behind Android Studio.

Obtaining Gradle

As with any build system, to use it, you need the build system's engine itself.

If you will only be using Gradle in the context of Android Studio, the IDE will take care of getting Gradle for you. If, however, you are planning on using Gradle outside of Android Studio (e.g., command-line builds), you will want to consider where your Gradle is coming from. This is particularly important for situations where you want to build the app with no IDE in sight, such as using a continuous integration (CI) server, like Jenkins.

Also, the way that Android Studio works with Gradle — called the Gradle Wrapper – opens up security issues for your development machine, if you are the sort to download open source projects from places like GitHub and try using them.

Direct Installation

What most developers looking to use Gradle outside of Android Studio will wind up doing is installing Gradle directly.

The Gradle download page contains links to ZIP archives for Gradle itself: binaries, source code, or both.

You can unZIP this archive to your desired location on your development machine.

Linux Packages

You may be able to obtain Gradle via a package manager on Linux environments. For example, there is an Ubuntu PPA for Gradle.

The gradlew Wrapper

If you are starting from a project that somebody else has published, you may find a `gradlew` and `gradlew.bat` file in the project root, along with a `gradle/` directory. This represents the "Gradle Wrapper".

The Gradle Wrapper consists of three pieces:

- the batch file (`gradlew.bat`) or shell script (`gradlew`)
- the JAR file used by the batch file and shell script (in the `gradle/wrapper/` directory)
- the `gradle-wrapper.properties` file (also in the `gradle/wrapper/` directory)

Android Studio uses the `gradle-wrapper.properties` file to determine where to download Gradle from, for use in your project, from the `distributionUrl` property in that file:

```
#Sun Nov 05 06:56:08 EST 2017
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.1-all.zip
```

(from Basic/Button/gradle/wrapper/gradle-wrapper.properties)

When you create or import a project, or if you change the version of Gradle referenced in the properties file, Android Studio will download the Gradle pointed to by the `distributionUrl` property and install it to a `.gradle/` directory (note the leading `.`) in your project. That version of Gradle will be what Android Studio uses.

RULE #1: Only use a `distributionUrl` that you trust.

If you are importing an Android project from a third party — such as the samples for this book — and they contain the `gradle/wrapper/gradle-wrapper.properties` file, examine it to see where the `distributionUrl` is pointing to. If it is loading from `services.gradle.org`, or from an internal enterprise server, it is probably

trustworthy. If it is pointing to a URL located somewhere else, consider whether you really want to use that version of Gradle, considering that it may have been tampered with.

The batch file, shell script, and JAR file are there to support command-line builds. If you run the `gradlew` command, it will use a local copy of Gradle installed in `.gradle/` in the project. If there is no such copy of Gradle, `gradlew` will download Gradle from the `distributionUrl`, as does Android Studio. Note that Android Studio does not use `gradlew` for this role — that logic is built into Android Studio itself.

RULE #2: Only use a `gradlew` that you REALLY trust.

It is relatively easy to examine a `.properties` file to check a URL to see if it seems valid. Making sense of a batch file or shell script can be cumbersome. Decompiling a JAR file and making sense of it can be rather difficult. Yet, if you use `gradlew` that you obtained from somebody, that script and JAR are running on your development machine, as is the copy of Gradle that they install. If that code was tampered with, the malware has complete access to your development machine and anything that it can reach, such as servers within your organization.

Note that you do not have to use the Gradle Wrapper at all. If you would rather not worry about it, install a version of Gradle on your development machine yourself and remove the Gradle Wrapper files. You can use the `gradle` command to build your app (if your Gradle's `bin/` directory is in your `PATH`), and Android Studio will use your Gradle installation (if you teach it where to find it, such as via the `GRADLE_HOME` environment variable).

Versions of Gradle and the Android Gradle Plugin

The Android Gradle Plugin that we will use to give Gradle "super Android powers!" is updated periodically. Each update has its corresponding required version of Gradle. Google maintains a page listing the Gradle versions supported by each Android Gradle Plugin version

If you are using the Gradle Wrapper, you are using an installation of Gradle that is local to the project. So long as the version of Gradle in the project matches the version of the Android Gradle Plugin requested in the project's `build.gradle` file — as will be covered later in this chapter — you should be in fine shape.

If you are not using the Gradle Wrapper, you will need to decide when to take on a new Android Gradle Plugin release and plan to update your Gradle installation and `build.gradle` files in tandem at that point.

Gradle Environment Variables

If you installed Gradle yourself, you will want to define a `GRADLE_HOME` environment variable, pointing to where you installed Gradle, and to add the `bin/` directory inside of Gradle to your `PATH` environment variable.

You may also consider setting up a `GRADLE_USER_HOME` environment variable, pointing to a directory in which Gradle can create a `.gradle/` subdirectory, for per-user caches and related materials. By default, Gradle will use your standard home directory.

Examining the Gradle Files

An Android Studio project usually has two `build.gradle` files, one at the project level and one at the "module" level (e.g., in the `app/` directory).

The Project-Level File

The `build.gradle` file in the project directory controls the Gradle configuration for all modules in your project. Right now, most likely you only have one module, and many apps only ever use one module. However, it is possible for you to add other modules to this project, and we will explore reasons for doing so later in this book.

Here is a typical top-level `build.gradle` file:

```
// Top-level build file where you can add configuration options common to all
sub-projects/modules.

buildscript {

    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0'
```

```
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

buildscript

In Groovy terms, a "closure" is a block of code wrapped in braces (`{ }`).

The `buildscript` closure in Gradle is where you configure the JARs and such that Gradle itself will use for interpreting the rest of the file. Hence, here you are not configuring your *project* so much as you are configuring *the build itself*.

The `repositories` closure inside the `buildscript` closure indicates where plugins can come from. Here, `jcenter()` is a built-in method that teaches Gradle about JCenter, a popular location for obtaining open source libraries. Similarly, `google()` is a built-in method that teaches Gradle about a site where it can download plugins from Google.

The `dependencies` closure indicates what is required to be able to run the rest of the build script. `classpath 'com.android.tools.build:gradle:3.0.0'` is not especially well-documented by the Gradle team. However the `'com.android.tools.build:gradle:3.0.0'` portion means:

- Find the `com.android.tools.build` group of plugins
- Find the `gradle` artifact within that group
- Ensure that we have version `3.0.0` of the plugin

The first time you run your build, with the `buildscript` closure as shown above, Gradle will notice that you do not have this dependency. It will then download that artifact from Google, as Google serves up its plugin from its own site nowadays.

Sometimes, the last segment of the version is replaced with a `+` sign (e.g., `3.0.+`). This tells Gradle to download the latest version, thereby automatically upgrading you to the latest patch-level (e.g., `3.0.1` at some point).

allprojects

The `allprojects` closure says "apply these settings to all modules in this project". Here, we are setting up `jcenter()` and `google()` as places to find libraries used in any of the modules in our project. We will use lots of libraries in our projects — having these "repositories" set up in `allprojects` makes it simpler for us to request them.

The Module-Level Gradle File

In your `app/` module, you will also find a `build.gradle` file. This has settings unique for this module, independent of any other module that your project may have in the future.

Here is a typical module-level `build.gradle` file:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 26
    defaultConfig {
        applicationId "com.commonsware.myapplication"
        minSdkVersion 21
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
```

```
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

For now, let's focus on some key elements of this file.

dependencies

This `build.gradle` file also has a `dependencies` closure. Whereas the `dependencies` closure in the `buildscript` closure in the top-level `build.gradle` file is for libraries used by the build process, the `dependencies` closure in the module's `build.gradle` file is for libraries used by your code in that module. We will get into the concept of these libraries later in the book.

android

The `android` closure contains all of the Android-specific configuration information. This closure is what the Android plugin enables, where the plugin itself comes from the `apply plugin: 'com.android.application'` line at the top, coupled with the `classpath` line from the project-level `build.gradle` file.

But before we get into what is in this closure, we should "switch gears" and talk about the manifest file, as what goes in the `android` closure is related to what goes in the manifest file.

Introducing the Manifest

The foundation for any Android application is the manifest file: `AndroidManifest.xml`. This will be in your `app` module's `src/main/` directory (the `main` source set) for typical Android Studio projects.

Here is where you declare what is inside your application — the activities, the services, and so on. You also indicate how these pieces attach themselves to the overall Android system; for example, you indicate which activity (or activities) should appear on the device's main menu (a.k.a., launcher).

When you create your application, you will get a starter manifest generated for you. For a simple application, offering a single activity and nothing else, the auto-generated manifest will probably work out fine, or perhaps require a few minor

modifications. On the other end of the spectrum, the manifest file for the Android API demo suite is over 1,000 lines long. Your production Android applications will probably fall somewhere in the middle.

As mentioned previously, some items can be defined in both the manifest and in a `build.gradle` file. The approach of putting that stuff in the manifest still works. For Android Studio users, you will probably use the Gradle file and not have those common elements be defined in the manifest.

Things In Common Between the Manifest and Gradle

There are a few key items that can be defined in the manifest and can be overridden in `build.gradle` statements. These items are fairly important to the development and operation of our Android apps as well.

Package Name and Application ID

The root of all manifest files is, not surprisingly, a `manifest` element:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.empublite">
```

Note the `android` namespace declaration. You will only use the namespace on many of the attributes, not the elements (e.g., `<manifest>`, not `<android:manifest>`).

The biggest piece of information you need to supply on the `<manifest>` element is the `package` attribute.

The `package` attribute will always need to be in the manifest, even for Android Studio projects. The `package` attribute will control where some source code is generated for us, notably some `R` and `BuildConfig` classes that we will encounter later in the book.

Since the `package` value is used for Java code generation, it has to be a valid Java package name. Java convention says that the package name should be based on a reverse domain name (e.g., `com.commonsware.myapplication`), where you own the domain in question. That way, it is unlikely that anyone else will accidentally collide with the same name.

The `package` also serves as our app's default "application ID". This needs to be a unique identifier, such that:

- no two apps can be installed on the same device at the same time with the same application ID
- no two apps can be uploaded to the Play Store with the same application ID (and other distribution channels may have the same limitation)

By default, the application ID is the `package` value, but Android Studio users can override it in their Gradle build files. Specifically, inside of the `android` closure can be a `defaultConfig` closure, and inside of there can be an `applicationId` statement:

```
android {
    // other stuff

    defaultConfig {
        applicationId "com.commonsware.myapplication"
        // more other stuff
    }
}
```

Not only can Android Studio users override the application ID in the `defaultConfig` closure, but there are ways of having different application ID values for different scenarios, such as a debug build versus a release build. We will explore that more later in the book.

minSdkVersion and targetSdkVersion

Your `defaultConfig` closure inside the `android` closure in your module's `build.gradle` file has a pair of properties named `minSdkVersion` and `targetSdkVersion`. Technically, these override values that could be defined via a `<uses-sdk>` element in the manifest, though few projects will have such an element nowadays.

Of the two, the more critical one is `minSdkVersion`. This indicates what is the oldest version of Android you are testing with your application. The value of the attribute is an integer representing the Android API level. So, if you are only testing your application on Android 4.1 and newer versions of Android, you would set your `minSdkVersion` to be `16`. The initial value is what you requested when you had Android Studio create the project.

You can also specify a `targetSdkVersion`. This indicates what version of Android you are thinking of as you are writing your code. If your application is run on a newer version of Android, Android may do some things to try to improve compatibility of your code with respect to changes made in the newer Android. Nowadays, most Android developers should specify a target SDK version of `15` or higher. We will start to explore more about the `targetSdkVersion` as we get deeper into the book; for the moment, whatever your IDE gives you as a default value is probably a fine starting point.

Version Code and Version Name

Similarly, the `defaultConfig` closure has `versionCode` and `versionName` properties. In principle, these override `android:versionName` and `android:versionCode` attributes on the root `<manifest>` element in the manifest, though you will not find many projects using those XML attributes.

These two values represent the versions of your application. The `versionName` value is what the user will see for a version indicator in the Applications details screen for your app in their Settings application. Also, the version name is used by the Play Store listing, if you are distributing your application that way. The version name can be any string value you want.

The `versionCode`, on the other hand, must be an integer, and newer versions must have higher version codes than do older versions. Android and the Play Store will compare the version code of a new APK to the version code of an installed application to determine if the new APK is indeed an update. The typical approach is to start the version code at `1` and increment it with each production release of your application, though you can choose another convention if you wish. During development, you can leave these alone, but when you move to production, these attributes will matter greatly.

Other Gradle Items of Note

The `android` closure has a `compileSdkVersion` property. `compileSdkVersion` specifies the API level to be compiled against, usually as a simple API level integer (e.g., 19). This indicates what Java classes, methods, and so on are available to you when you write your app. Typically, you set this to be the latest production release of Android, using the values noted earlier in the book.

The `android` closure *may* have a `buildToolsVersion` property. `buildToolsVersion` indicates the version of the Android SDK build tools that you wish to use with this project. The Android Gradle Plugin really is a thin wrapper around a series of "build tools" that handle most of the work of creating an APK out of your project. If your `android` closure does not have `buildToolsVersion`, the Android Gradle Plugin will use its own default version of these build tools, and for many projects this will suffice. In this book, most projects will state their `buildToolsVersion`, though some will skip that line and use the plugin-provided default.

Where's the GUI?

You might wonder why we have to slog through all of this Groovy code and wonder if there is some GUI for affecting Gradle settings.

The answer is yes... and no.

There is the project structure dialog, that allows you to maintain some of this stuff. And you are welcome to try it. However, the more complex your build becomes, the more likely it is that the GUI will not suffice, and you will need to work with the Gradle build files more directly. Hence, this book will tend to focus on the build files.

The Rest of the Manifest

Not everything in the manifest can be overridden in the Gradle build files. Here are a few key items that will always be defined in the manifest, not in a `build.gradle` file.

An Application For Your Application

In your initial project's manifest, the primary child of the `<manifest>` element is an `<application>` element.

By default, when you create a new Android project, you get a single `<activity>` element inside the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.myapplication"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <application
```

```
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name="MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

The `<activity>` element supplies `android:name` for the class implementing the activity, `android:label` for the display name of the activity, and (sometimes) an `<intent-filter>` child element describing under what conditions this activity will be displayed. The stock `<activity>` element sets up your activity to appear in the launcher, so users can choose to run it. As we'll see later in this book, you can have several activities in one project, if you so choose.

The `android:name` attribute, in this case, has a bare Java class name (`MainActivity`). Sometimes, you will see `android:name` with a fully-qualified class name (e.g., `com.commonsware.myapplication.MainActivity`). Sometimes, you will see a Java class name with a single dot as a prefix (e.g., `.MainActivity`). Both `MainActivity` and `.MainActivity` refer to a Java class that will be in your project's package — the one you declared in the `package` attribute of the `<manifest>` element.

Supporting Multiple Screens

Android devices come with a wide range of screen sizes, from 2.8" tiny smartphones to 46" TVs. Android divides these into four buckets, based on physical size and the distance at which they are usually viewed:

1. Small (under 3")
2. Normal (3" to around 4.5")
3. Large (4.5" to around 10")
4. Extra-large (over 10")

By default, your application will support small and normal screens. It also will support large and extra-large screens via some automated conversion code built into Android.

To truly support all the screen sizes you want, you should consider adding a `<supports-screens>` element to your manifest. This enumerates the screen sizes you have explicit support for. For example, if you are providing custom UI support for large or extra-large screens, you will want to have the `<supports-screens>` element. So, while the starting manifest file works, handling multiple screen sizes is something you will want to think about.

You wind up with an element akin to:

```
<supports-screens
  android:largeScreens="true"
  android:normalScreens="true"
  android:smallScreens="false"
  android:xlargeScreens="true" />
```

Much more information about providing solid support for all screen sizes, including samples of the `<supports-screens>` element, will be found later in this book as we cover large-screen strategies.

Other Stuff

As we proceed through the book, you will find other elements being added to the manifest, such as:

- `<uses-permission>`, to tell the user that you need permission to use certain device capabilities, such as accessing the Internet
- `<uses-feature>`, to tell Android that you need the device to have certain features (e.g., a camera), and therefore your app should not be installed on devices lacking such features
- `<meta-data>`, for bits of information needed by particular extensions to Android, such as by `FileProvider`.

These and other elements will be introduced elsewhere in the book.

Learning More About Gradle

This book will go into more about Gradle, both in the core chapters and in the trails. But, the focus will be on the Android Gradle Plugin, and Gradle itself offers a lot more than that. The Gradle Web site hosts documentation, links to Gradle-specific books, and links to other Gradle educational resources.

Visit the Trails!

There are a few more chapters in this book getting into more details about the use of Gradle and the Android Gradle Plugin.

- Gradle and Tasks explains how we ask Gradle to do things on our behalf ("tasks"), such as compile our APK for us
- Gradle and Build Variants gets into what capabilities we get from the Gradle project structure, including the ability to configure "build types" and "product flavors"

There is also the "Advanced Gradle for Android Tips" chapter for other Gradle topics, and the chapter on manifest merging in Gradle.

Tutorial #3 - Manifest Changes

As we build `EmPubLite`, we will need to make a number of changes to our project's manifest. In this tutorial, we will take care of a couple of these changes, to show you how to manipulate the `AndroidManifest.xml` file. Future tutorials will make yet more changes.

Android Studio users will also get their first chance to work with the `build.gradle` file.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Some Notes About Relative Paths

In these tutorials, you will see references to relative paths, like `AndroidManifest.xml`, `res/layout/`, and so on.

You should interpret these paths as being relative to the `app/src/main/` directory within the project, except as otherwise noted. So, for example, Step #1 below will ask you to open `AndroidManifest.xml` — that file can be found in `app/src/main/AndroidManifest.xml` from the project root.

Step #1: Supporting Screens

Our application will restrict its supported screen sizes. Tablets make for ideal ebook readers. Phones can also be used, but the smaller the phone, the more difficult it

will be to come up with a UI that will let the user do everything that is needed, yet still have room for more than a sentence or two of the book at a time.

We will get into screen size strategies and their details later in this book. For the moment, though, we will add a `<supports-screens>` element to keep our application off "small" screen devices (under 3" diagonal size).

Android Studio users can double-click on `AndroidManifest.xml` in the project explorer.

As a child of the root `<manifest>` element, add a `<supports-screens>` element as follows:

```
<supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>
```

Step #2: Blocking Backups

If you look at the `<application>` element, you will see that it has a few attributes, including `android:allowBackup="true"`. This attribute indicates that EmPubLite should participate in Android's automatic backup system.

That is not a good idea, until you understand the technical and legal ramifications of that choice, which we will explore much later in this book.

In the short term, change `android:allowBackup` to be `false`.

Step #3: Ignoring Lint

Even after that change, the `application` element name may have a beige background. If you hover your mouse over it and look at the explanatory tooltip, you will see that it is complaining that this app is not indexable, and that you should add an `ACTION_VIEW` activity to the app.

This is ridiculous. This app (hopefully) will never wind up on the Play Store, and so Google's "app indexing" capability will never be relevant.

Put your text cursor somewhere inside the `application` element name and press Alt-Enter (or Option-Return on macOS). This should bring up a popup window showing some "quick fixes" for the problem:

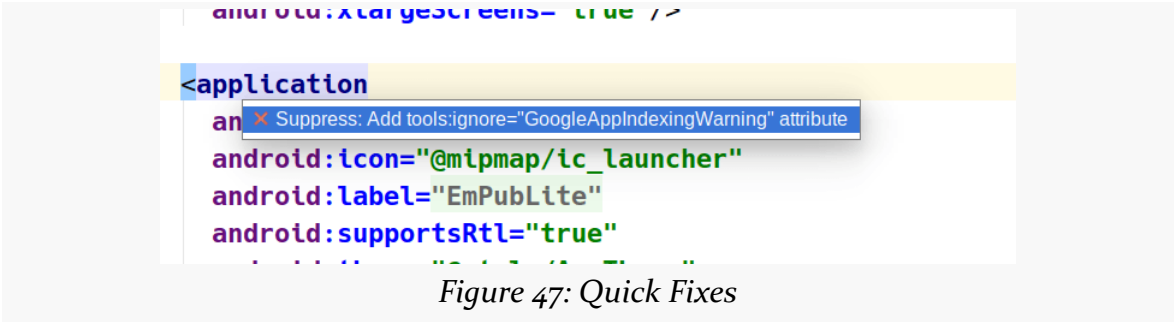


Figure 47: Quick Fixes

Choose the "suppress" option. Then, press Ctrl-Alt-L (or Command-Option-L on macOS) to reformat the file. You will wind up with something like:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.empublite"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true" />

  <application
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:ignore="GoogleAppIndexingWarning">
    <activity android:name=".EmPubLiteActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from EmPubLite-AndroidStudio/T3-Manifest/EmPubLite/app/src/main/AndroidManifest.xml)

The `<application>` element now has a `tools:ignore="GoogleAppIndexingWarning"` attribute, and the root `<manifest>` element defines the `tools` XML namespace. The net effect is that we are telling the build tools — specifically the Lint utility – that it should ignore this particular issue.

In Our Next Episode…

… we will make some changes to the resources of our tutorial project

Some Words About Resources

It is quite likely that by this point in time, you are "chomping at the bit" to get into actually writing some code. This is understandable. That being said, before we dive into the Java source code for our stub project, we really should chat briefly about resources.

Resources are static bits of information held outside the Java source code. As we discussed previously, resources are stored as files under the `res/` directory in your source set (e.g., `app/src/main/res/`). Here is where you will find all your icons and other images, your externalized strings for internationalization, and more.

These are separate from the Java source code not only because they are different in format. They are separate because you can have *multiple* definitions of a resource, to use in different circumstances. For example, with internationalization, you will have strings for different languages. Your Java code will be able to remain largely oblivious to this, as Android will choose the right resource to use, from all candidates, in a given circumstance (e.g., choose the Spanish string if the device's locale is set to Spanish).

We will cover all the details of these resource sets later in the book. Right now, we need to discuss the resources in use by our stub project, plus one more.

This chapter will refer to the `res/` directory as a shorthand for the `app/src/main/res/` directory of the project.

String Theory

Keeping your labels and other bits of text outside the main source code of your application is generally considered to be a very good idea. In particular, it helps with

internationalization (I18N) and localization (L10N). Even if you are not going to translate your strings to other languages, it is easier to make corrections if all the strings are in one spot instead of scattered throughout your source code.

Plain Strings

Generally speaking, all you need to do is have an XML file in the `res/values` directory (typically named `res/values/strings.xml`), with a `resources` root element, and one child `string` element for each string you wish to encode as a resource. The `string` element takes a `name` attribute, which is the unique name for this string, and a single text element containing the text of the string:

```
<resources>
  <string name="quick">The quick brown fox...</string>
  <string name="laughs">He who laughs last...</string>
</resources>
```

One tricky part is if the string value contains a quote or an apostrophe. In those cases, you will want to escape those values, by preceding them with a backslash (e.g., `These are the times that try men\'s souls`). Or, if it is just an apostrophe, you could enclose the value in quotes (e.g., `"These are the times that try men's souls."`).

For example, a project's `strings.xml` file could look like this:

```
<resources>
  <string name="app_name">EmPubLite</string>
</resources>
```

(from EmPubLite-AndroidStudio/T3-Manifest/EmPubLite/app/src/main/res/values/strings.xml)

We can reference these string resources from various locations, in our Java source code and elsewhere. For example, the `app_name` string resource often is used in the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.empublite"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
```

```
    android:xlargeScreens="true" />

  <application
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:ignore="GoogleAppIndexingWarning">
    <activity android:name=".EmPubLiteActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from EmPubLite-AndroidStudio/T3-Manifest/EmPubLite/app/src/main/AndroidManifest.xml)

Here, the `android:label` attribute of the `<application>` element refers to the `app_name` string resource. This will appear in a few places in the application, notably in the list of installed applications in Settings. So, if you wish to change how your application's name appears in these places, simply adjust the `app_name` string resource to suit.

The syntax `@string/app_name` tells Android "find the string resource named `app_name`". This causes Android to scan the appropriate `strings.xml` file (or any other file containing string resources in your `res/values/` directory) to try to find `app_name`.

Um, Wait, My Manifest Does Not Look Like That

When you view a manifest like that in Android Studio, it may appear as though you are not using resources, as you may not see `@string/...` references:

```
<application
  android:allowBackup="false"
  android:icon="@mipmap/ic_launcher"
  android:label="EmPubLite"
  android:supportsRtl="true"
  android:theme="@style/AppTheme"
  tools:ignore="GoogleAppIndexingWarning">
  <activity android:name=".EmPubLiteActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />

      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

Figure 48: AndroidManifest.xml, As Initially Viewed in Android Studio

Here, `android:label` looks as though it is the hard-coded value "EmPubLite".

However, notice that the attribute value is formatted differently than the others. The rest are green text with a white background, while this one is gray text with a shaded background.

That is because Android Studio is lying to you.

If you hover your mouse over the value, you will see the *real* attribute appear just below it:

```
<application
  android:allowBackup="false"
  android:icon="@mipmap/ic_launcher"
  android:label="EmPubLite"
  android:label="@string/app_name"
  android:theme="@style/AppTheme"
  tools:ignore="GoogleAppIndexingWarning">
  <activity android:name=".EmPubLiteActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />

      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

Figure 49: AndroidManifest.xml, With Mouse Hovering Over "EmPubLite"

And, if you click on the fake value, you will see the real XML, with the real string resource value.

What is happening is that Android Studio, by default, will substitute a candidate value for the resource in its presentation of the manifest, other resources that refer to resources, and even Java code. Any time you see that gray-on-light-blue formatting, remember that this is not the real value, and that you have to uncover the real value via hovering over it or clicking on it.

Styled Text

Many things in Android can display rich text, where the text has been formatted using some lightweight HTML markup: `<b>`, `<i>`, and `<u>`. Your string resources support this, simply by using the HTML tags as you would in a Web page:

```
<resources>
  <string name="b">This has <b>bold</b> in it.</string>
  <string name="i">Whereas this has <i>italics</i>!</string>
</resources>
```

CDATA. CDATA Run. Run, DATA, Run.

Since a strings resource XML file is an XML file, if your message contains <, >, or & characters (other than the formatting tags listed above), you will need to use a `CDATA` section:

```
<string name="report_body">
<![CDATA[
<html>
<body>
<h1>TPS Report for: {{reportDate}}</h1>
<p>Here are the contents of the TPS report:</p>
<p>{{message}}</p>
<p>If you have any questions regarding this report, please
do <b>not</b> ask Mark Murphy.</p>
</body>
</html>
]]>
  </string>
```

The Directory Name

Our string resources in our stub project are in the `res/values/strings.xml` file. Since this directory name (`values`) has no suffixes, the string resources in that directory will be valid for any sort of situation, including any locale for the device. We will need additional directories, with distinct `strings.xml` files, to support other languages. We will cover how to do that later in this book.

Editing String Resources

If you double-click on a string resource file, like `res/values/strings.xml`, in Android Studio, you are presented the XML and edit it that way. There is an option for entering a dedicated string translation view, covered later in this book.

Multi-Locale Support

Android 7.0+ users can indicate that they support more than one language:

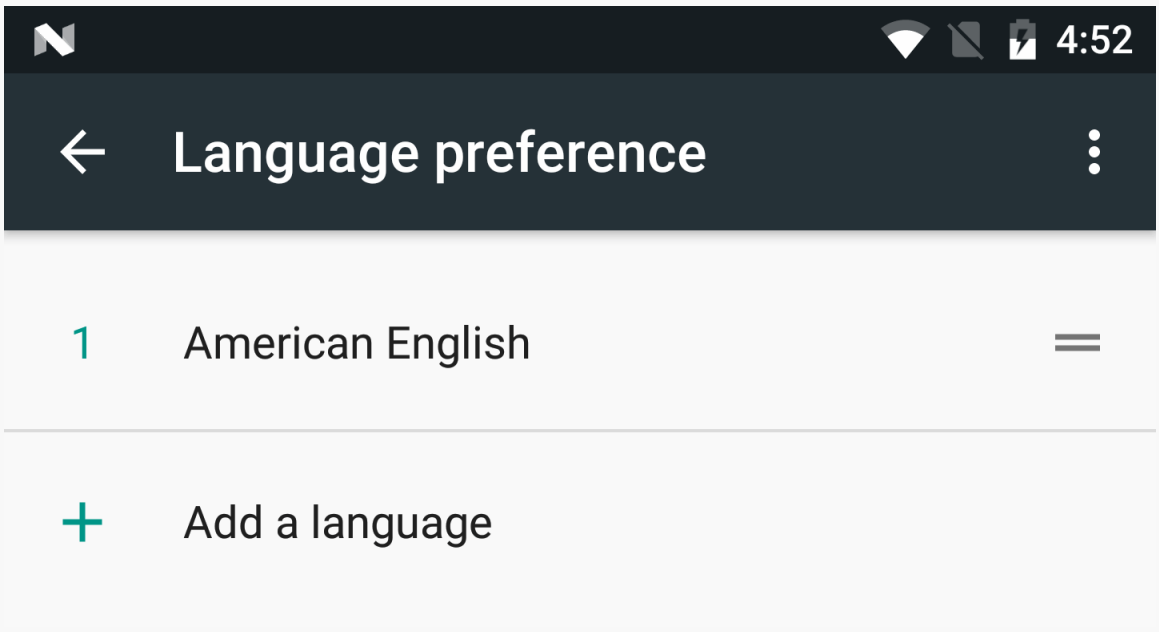


Figure 50: Android 7.0 Language Settings

The user can choose the relative priorities of these languages, by grabbing the handle on the right side of the row and dragging the language higher or lower in the list.

This has impacts on resource resolution for any locale-dependent resources, such as strings. Now Android will check multiple languages for resource matches, before falling back to the default language (e.g., whatever you have in `res/values/strings.xml`). Hence, it is important that you ensure that you have a complete set of strings for every language that you support, lest the user perhaps wind up with a mixed set of languages in the UI.

You can find out what languages the user has requested via a `LocaleList` class and its `getDefault()` `static` method. This, as the name suggests, has a list of `Locale` objects representing the user's preferred languages. If you had previously been using `Locale` alone for this (e.g., for specialized in-app language assistance beyond resources), you will want to switch to `LocaleList` for Android 7.0 and beyond.

Got the Picture?

All Android versions support images in the PNG, JPEG, and GIF formats. GIF is officially discouraged, however; PNG is the overall preferred format. Android also

supports some proprietary XML-based image formats, though we will not discuss those at length until later in the book. Many newer versions of Android also support Google's WebP image format, though this is not especially popular.

There are two types of resources that use images like these: drawables and mipmaps. In truth, they are nearly identical. Mipmaps are used mostly for "launcher icons" — the icons seen in home screen launchers that identify activities that the user can start. Drawables hold everything else.

(if you are a seasoned Android developer and are reading this section: while drawable resources might be removed when packaging an APK, such as for the Android Gradle Plugin `split` system for making density-specific editions of an app, mipmap resources are left alone, apparently)

It is possible to have `res/drawable/` and `res/mipmap/` directories in an Android module. However, you will not find bitmaps there usually. Instead, those reside in directories like `res/drawable-mdpi/` and `res/drawable-hdpi/`.

These refer to distinct resource sets. The suffixes (e.g., `-mdpi`, `-hdpi`) are filters, indicating under what circumstances the images stored in those directories should be used. Specifically, `-ldpi` indicates images that should be used on devices with low-density screens (around 120 dots-per-inch, or "dpi"). The `-mdpi` suffix indicates resources for medium-density screens (around 160dpi), `-hdpi` indicates resources for high-density screens (around 240dpi). `-xhdpi` indicates resources for extra-high-density screens (around 320dpi), `-xxhdpi` indicates extra-extra-high-density screens (around 480dpi), `-xxxhdpi` indicates extra-extra-extra-high-density screens (around 640dpi), and so on.

In the `EmPubLite` tutorial project, you will find a series of `mipmap` directories with the same sorts of suffixes (e.g., `res/mipmap-hdpi`). Inside each of those directories, you will see an `ic_launcher.png` file. This is the stock icon that will be used for your application in the home screen launcher. Each of the images is of the same icon, but the higher-density icons have more pixels. The objective is for the image to be roughly the same physical size on every device, using higher densities to have more detailed images.

Our `AndroidManifest.xml` file then references our `ic_launcher` icon in the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.empublite"
```

```
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true" />

  <application
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:ignore="GoogleAppIndexingWarning">
    <activity android:name=".EmPubLiteActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from EmPubLite-AndroidStudio/T3-Manifest/EmPubLite/app/src/main/AndroidManifest.xml)

Note that the manifest simply refers to `@mipmap/ic_launcher`, telling Android to find a mipmap resource named `ic_launcher`. The resource reference does not indicate the file type of the resource — there is no `.png` in the resource identifier. This means you cannot have `ic_launcher.png` and `ic_launcher.jpg` in the same project, as they would both be identified by the same identifier. You will need to keep the "base name" (filename sans extension) distinct for all of your images.

Also, the `@mipmap/ic_launcher` reference does not mention what screen density to use. That is because *Android* will choose the right screen density to use, based upon the device that is running your app. You do not have to worry about it explicitly, beyond having multiple copies of your icon. If Android detects that the device has a screen density for which you lack an icon, Android will take the next-closest one and scale it.

Some Notes About WebP

Android 4.0 added partial support for Google's WebP image format, and Android 4.3 devices support the previously-missing features (lossless compression and transparency). WebP serves as a replacement for both PNG and JPEG, and in some circumstances it can result in smaller on-disk sizes for near-equivalent image quality.

Android Studio, starting with version 2.3, has special support to help you convert drawable resources and other images from JPEG and PNG to WebP. Simply right-click over the image in the project tree and choose "Convert to WebP" from the context menu.

Initially, you are given a window for controlling the quality and output:

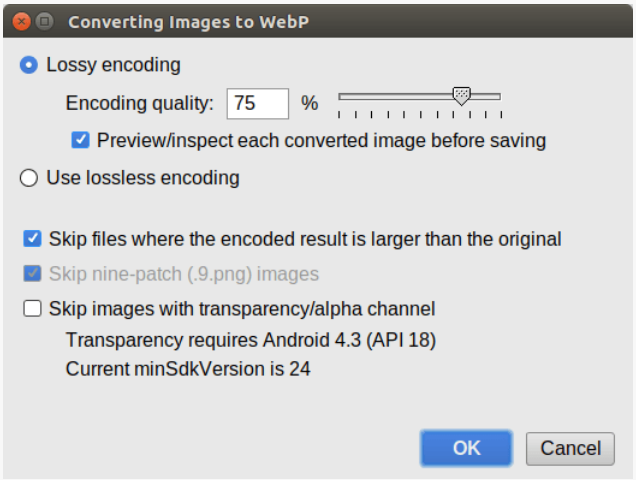


Figure 51: WebP Converter in Android Studio

"Lossy encoding" refers to the type performed by JPEG, taking into account that humans have limited ability to distinguish similar colors to achieve tighter compression. "Lossless encoding" refers to the type performed by PNG, where the compressed image is identical to the original, just as a ZIP file's contents are identical to the files before they were ZIPped. For lossy encoding, you can choose a quality percentage, where higher quality images will not compress as well.

You can also:

- Skip anything that results in a *bigger* image (as sometimes WebP will be bigger than the JPEG or PNG equivalent)

- Skip a type of PNG called a nine-patch PNG, used for widget backgrounds
- Skip images that use transparency, in case your `minSdkVersion` will not support such images

If you choose lossy compression and leave the "preview" checkbox checked, you are then presented with a window showing the results of the conversion at your requested quality level:

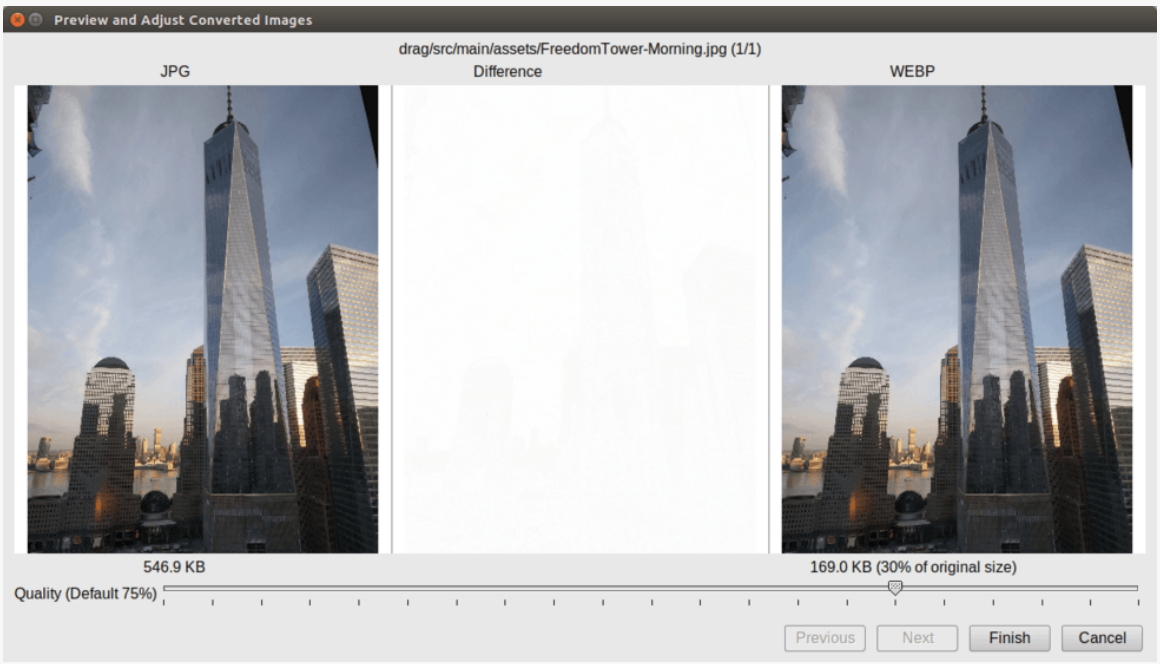


Figure 52: WebP Conversion Preview

You can adjust the quality slider below the images to see how the image changes with different quality levels and how much additional disk savings you will get from the WebP conversion.

When you are done, the WebP converter will replace your old PNG or JPEG file with the converted WebP image.

Dimensions

Dimensions are used in several places in Android to describe distances, such as a widget's size. There are several different units of measurement available to you:

1. `px` means hardware pixels, whose size will vary by device, since not all devices have the same screen density
2. `in` and `mm` for inches and millimeters, respectively, based on the actual size of the screen
3. `pt` for points, which in publishing terms is 1/72nd of an inch (again, based on the actual physical size of the screen)
4. `dip` (or `dp`) for density-independent pixels — one `dip` equals one hardware pixel for a ~160dpi resolution screen, but one `dip` equals two hardware pixels on a ~320dpi screen
5. `sp` for scaled pixels, where one `sp` equals one `dip` for normal font scale levels, increasing and decreasing as needed based upon the user's chosen font scale level in Settings

Dimension resources, by default, are held in a `dimens.xml` file in the `res/values/` directory that also holds your strings.

To encode a dimension as a resource, add a `dimen` element to `dimens.xml`, with a `name` attribute for your unique name for this resource, and a single child text element representing the value:

```
<resources>
  <dimen name="thin">10dip</dimen>
  <dimen name="fat">1in</dimen>
</resources>
```

In a layout, you can reference dimensions as `@dimen/...`, where the ellipsis is a placeholder for your unique name for the resource (e.g., `thin` and `fat` from the sample above). In Java, you reference dimension resources by the unique name prefixed with `R.dimen.` (e.g., `Resources.getDimension(R.dimen.thin)`).

While our stub project does not use dimension resources, we will be seeing them soon enough.

Editing Dimension Resources

As with most types of XML resources, Android Studio just has you edit the XML directly, when you double-click on the resource in the project explorer.

The Resource That Shall Not Be Named… Yet

Your stub project also has a `res/layout/` directory, in addition to the ones described above. That is for UI layouts, describing what your user interface should look like. We will get into the details of that type of resource as we start examining our user interfaces in an upcoming chapter.

Icons

Icons are `drawable` resources. Except when they are `mipmap` resources. And except when they are multiple resources, combined together at runtime to create an image with a "squircle" background. And...

Are you confused yet?

In theory, setting up icons for your app would be quite simple, and for years that was the case. Nowadays, setting up an app icon is unnecessarily complex, though at least Android Studio has an Asset Studio tool to try to make it a bit simpler.

In this chapter, we will explore what it takes to set up one of these app icons.

App Icons… And Everything Else

Each app has an icon. This is used for places like the list of installed apps in Settings. Traditionally — and to reduce user confusion — this icon is also used for the home screen launcher.

Google has been making this icon increasingly complicated over the past few years:

- App icons are `mipmap` resources, while everything else is a `drawable` resource
- Android 7.1 introduced the concept of a separate "round icon" that an app can have, which would be used in place of the regular app icon on certain Android 7.1 devices... then dropped this feature with Android 8.0
- Android 8.0 introduced the concept of "adaptive icons", where you have to provide separate "foreground" and "background" images, mostly so that certain home screen launchers can shape the background image as they want (square, round, "squircle", etc.)

Most other icons — other than notification icons – are comparatively straightforward.

Creating an App Icon with the Asset Studio

If you right-click over pretty much any directory in the Android Studio tree, the context menu will have an "Image Asset" option. This is also available from File > New > Image Asset. This brings up the Asset Studio, to help you to assemble icons.

By default, the Asset Studio has its "Icon Type" drop-down set for "Launcher Icons (Adaptive and Legacy)", which is how you set up an app icon nowadays:

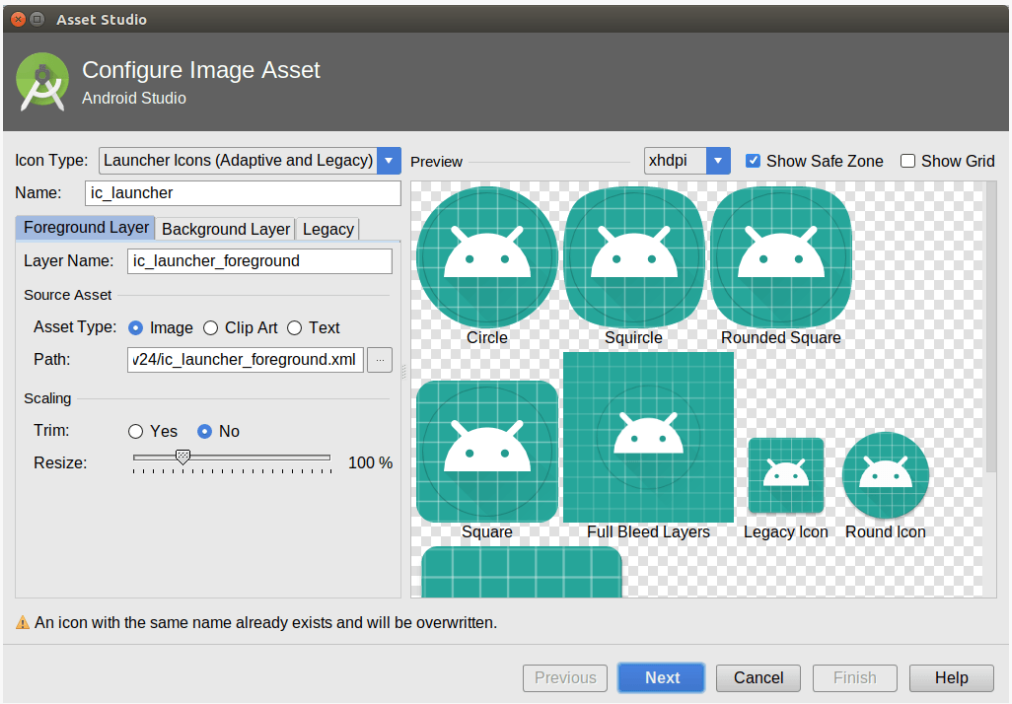


Figure 53: Asset Studio, As Initially Launched

The overall name for your launcher icon is found in the Name field, above the tabs. The default is `ic_launcher`, and unless you have a good reason to change it, you are best served by leaving it alone.

Foreground Layer

What you would ordinarily think of as your icon is what Android Studio and Android 8.0+ refer to as the "Foreground Layer". The Asset Studio starts on the Foreground Layer tab for you to configure this layer.

Your icon can come from three main sources:

- Some image of your own design, which you would load into the Foreground Layer tab by selecting the Image radio button, then clicking the "..." button next to the Path field, to browse your development machine and find that image
- A piece of canned clip art, which you would choose by clicking on the "Clip Art" radio button, clicking the "Clip Art" button to choose the image, and clicking the Color button to choose a color to apply to that image:

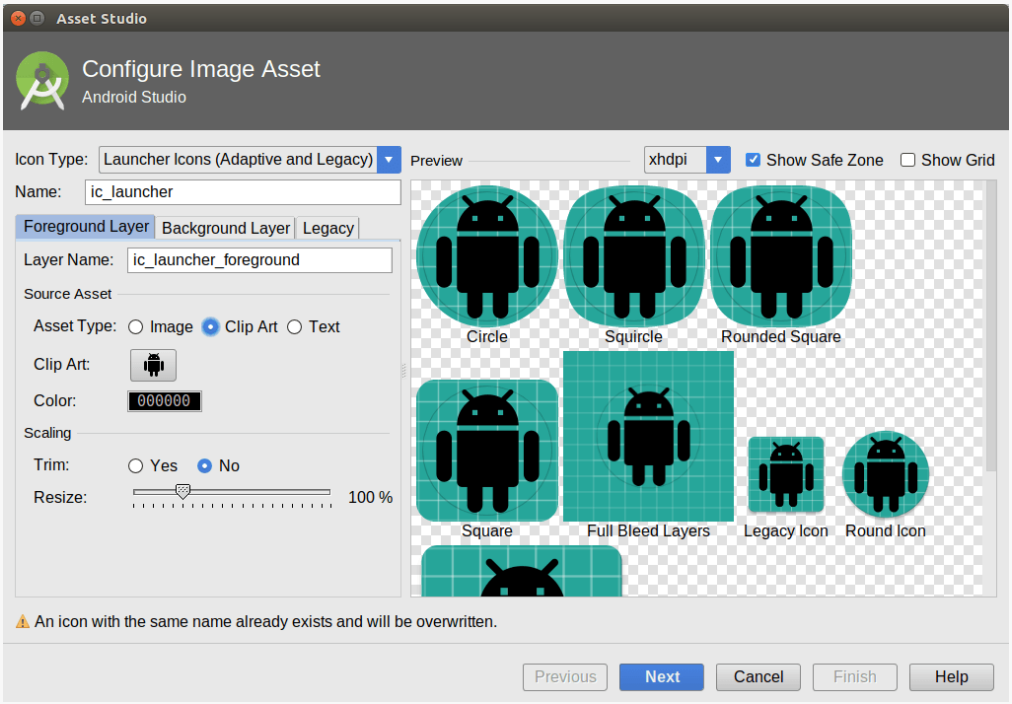


Figure 54: Asset Studio, Showing Clip Art Options

- A letter or two, which you would choose by clicking on the Text radio button, filling in 1-2 letters in the Text field, choosing a font from your development machine in the adjacent drop-down, and clicking the Color button to choose a color to apply to the font:

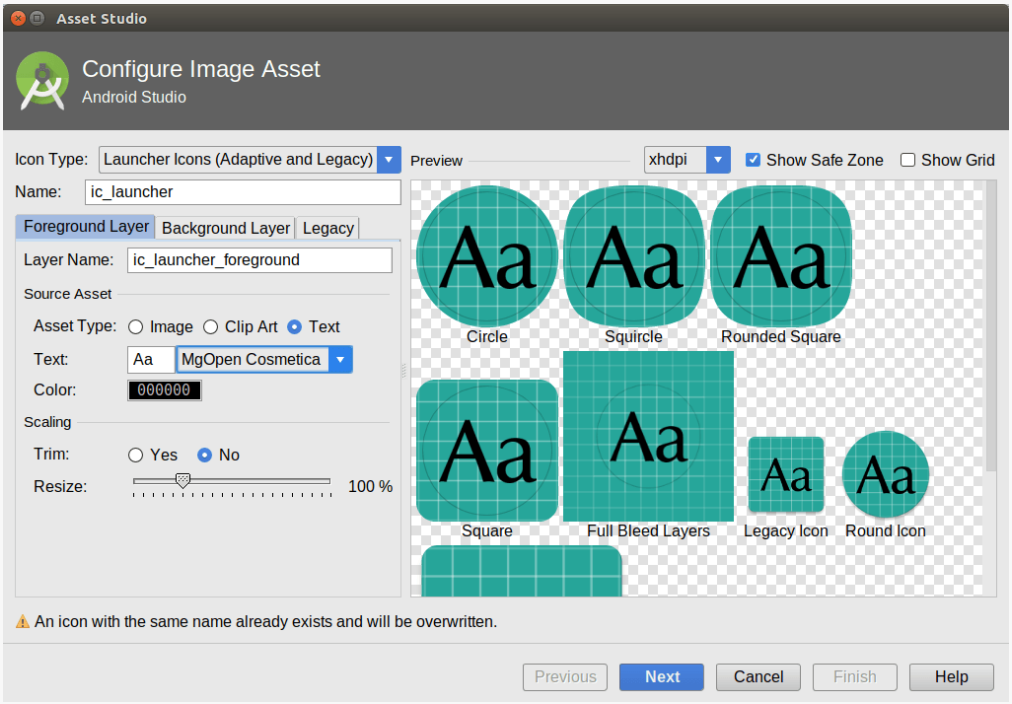


Figure 55: Asset Studio, Showing Text Options

For all three of these, you can:

- Choose the name to use for this image, in the “Layer Name” field
- Choose, via the Trim radio buttons, whether to remove all transparent space from the edges of the image (“Yes”) or not (“No”)
- Resize the image from its default size

That latter choice is particularly important, as you need to keep your foreground layer content within the “safe zone”. That shows up in the previews as a dark gray circle. So long as your foreground layer is in the safe zone, you should not have to worry about any launcher accidentally cutting off part of the layer when it renders your app icon.

Background Layer

In the preview area, by default, you will see a blue-green grid behind your foreground layer. That is the default background layer. If you would like to use something else, click the “Background Layer” tab:

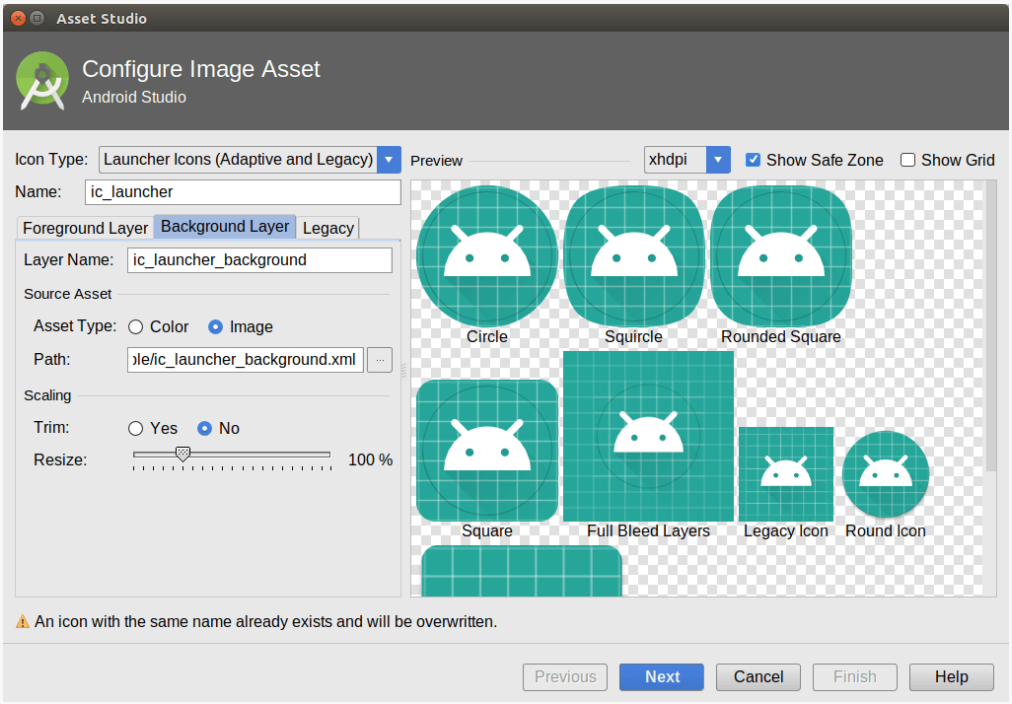


Figure 56: Asset Studio, Showing Background Layer

Your two main options are:

- a flat color
- an image of your choosing (akin to the foreground, where you click the "..." button next to the Path field to pick the background image)

The background needs to be designed to be cropped into a variety of shapes, such as those shown in the preview (circle, various rounded forms of squares, etc.). Hence, outside of flat colors, typical backgrounds will be gradients or simple patterns, such as the default grid.

Legacy

On Android 8.0+ devices, the foreground layer, background layer, and launcher-chosen shape (e.g., squircle) combine to create your app icon.

On older devices, the app icon is whatever you choose it to be, where the Legacy tab helps you decide what that is:

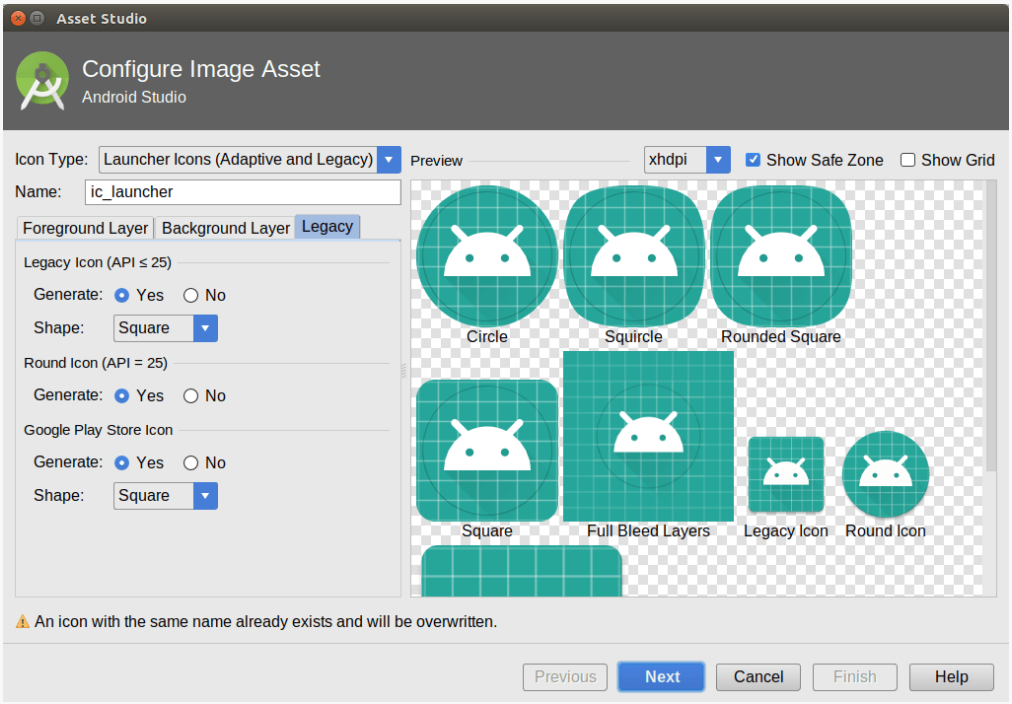


Figure 57: Asset Studio, Showing Legacy Options

The most important legacy option is the "Legacy Icon (API <= 25)" section. If your `minSdkVersion` is below 26, this will be the icon rendition that will be used as your app icon by default. The Asset Studio will merge your foreground and background layers itself, then apply your selected shape from the drop-down (e.g., square).

For Android 7.1 devices, you can also opt to have the Asset Studio create a separate round icon, that you can declare in your manifest, as will be seen later in this chapter.

If you are going to be distributing your app through the Play Store, you can also generate a Play Store rendition of your icon. This is reminiscent of the legacy icon, but at a higher resolution.

Generating the Icon

Once you have adjusted your app icon via the three tabs, click Next at the bottom of the Asset Studio wizard. This will bring up the final wizard page, showing you what will be generated for you by the wizard:

The Output Directories tree shows you each file that will be created or replaced. Those that show up in red are ones that will be replaced; ones that show up in black are new files. Typically, unless you changed the icon or layer names, most of the files will be replacements for files that exist already.

Some of these files will be typical bitmap-style resources. Some are vector drawables.

Using In Your Manifest

Then, in your manifest, you can have `android:icon` and optionally `android:roundIcon` attributes on the `<application>` element to associate your icons with your app. If you did not change the names of the icons, then your manifest should already have the appropriate attribute values.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.myapplication"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Creating Other Icons with the Asset Studio

In principle, you can use the Asset Studio to create other types of icons, by choosing another type of icon from the drop down, such as “Action Bar and Tab Icons”.

In practice, the Asset Studio does not do much for you here, other than create multiple versions of your icon for different screen densities.

For most Android app developers, there are two other options:

1. Get icons at the right resolutions for different densities from your graphic designer, perhaps exported from Adobe Photoshop
2. Create vector icons, as will be covered in an upcoming chapter

Tutorial #4 - Adjusting Our Resources

Our `EmPubLite` project has some initial resources. However, the defaults are not what we want for the long term. So, in addition to adding new resources in future tutorials, we will fix the ones we already have in this tutorial.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Changing the Name

Our application shows up everywhere as "EmPubLite":

- In the title bar of our activity
- As the caption under our icon in the home screen launcher
- In the Application list in the Settings app
- And so on

We should change that to be "EmPub Lite", adding a space for easier reading, and to illustrate that this is a "lite" version of the full `EmPub` application.

Double-click on the `res/values/strings.xml` file in your project explorer.

In the XML editor for the string resources, you will find an element that looks like:

```
<string name="app_name">EmPubLite</string>
```

Change the text node in this element to `EmPub Lite`. Then save your changes, giving you something like:

```
<resources>
  <string name="app_name">EmPub Lite</string>
</resources>
```

Step #2: Changing the Icon

The build tools provide us with a stock icon to use for the launcher — the actual image used varies by Android tools release. However, we can change it to something else. For example, we could use the icon portion of the CommonsWare logo:

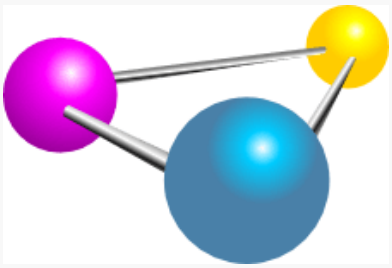


Figure 58: CommonsWare

Download the molecule PNG file from the CommonsWare Web site and save it somewhere on your development machine.

Then, right-click over the `res/` directory in your `main` source set in the project explorer, and choose New > Image Asset from the context menu. That will bring up the Asset Studio wizard:

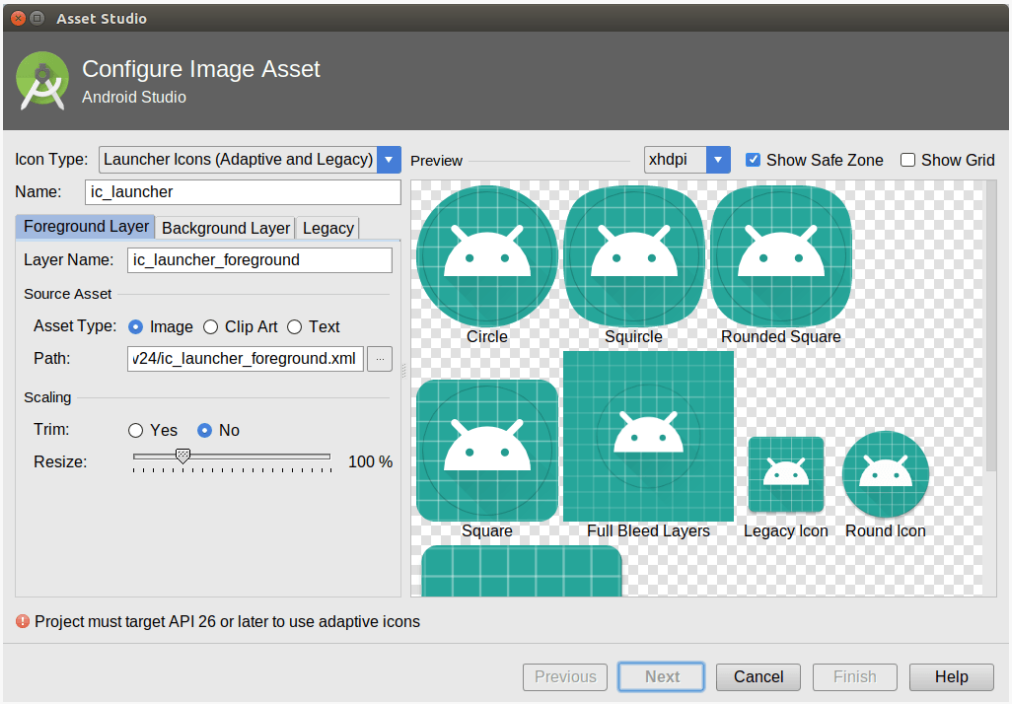


Figure 59: Asset Studio Wizard, First Page

In the Icon Type drop-down, choose “Launcher Icons (Legacy only)”. This will change the wizard to look like:

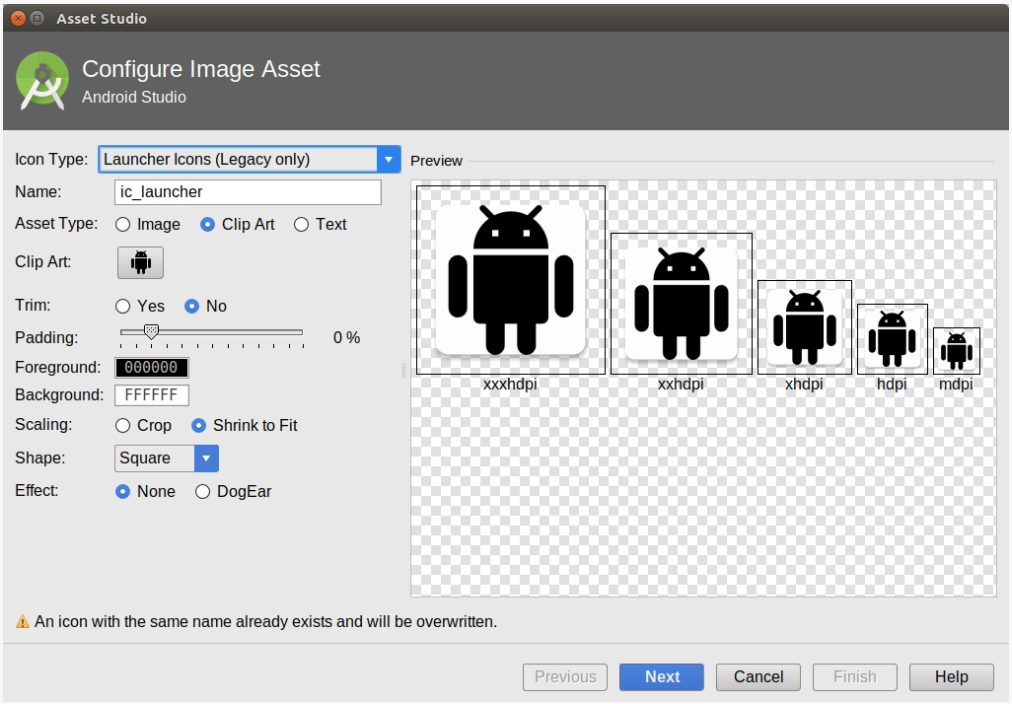


Figure 60: Asset Studio Wizard, First Page, Legacy Mode

Click the "Image" radio button in the "Asset Type" row. Then, click the "..." to the right of the "Path" field and choose the `molecule.png` file that you downloaded. Also, ensure that "Scaling" is set to "Shrink to Fit", and choose "None" from the "Shape" drop-down. This should give you a preview of what the icons will look like:

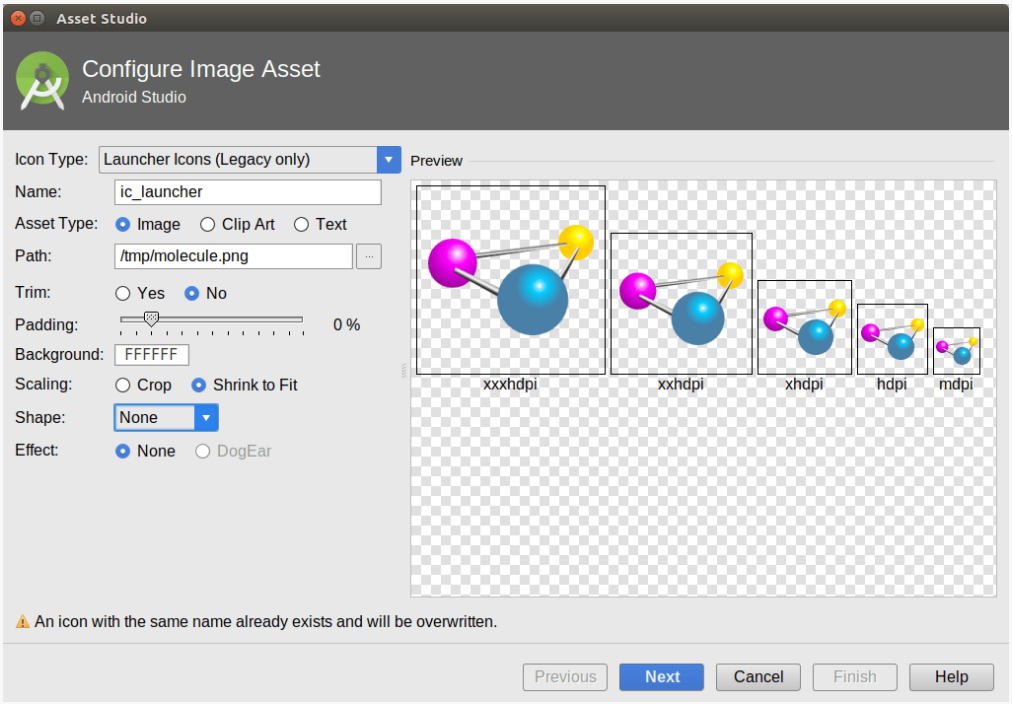


Figure 61: Asset Studio Wizard, First Page, After Loading Image

Leave the rest of the wizard alone, then click Next to proceed to the next page:

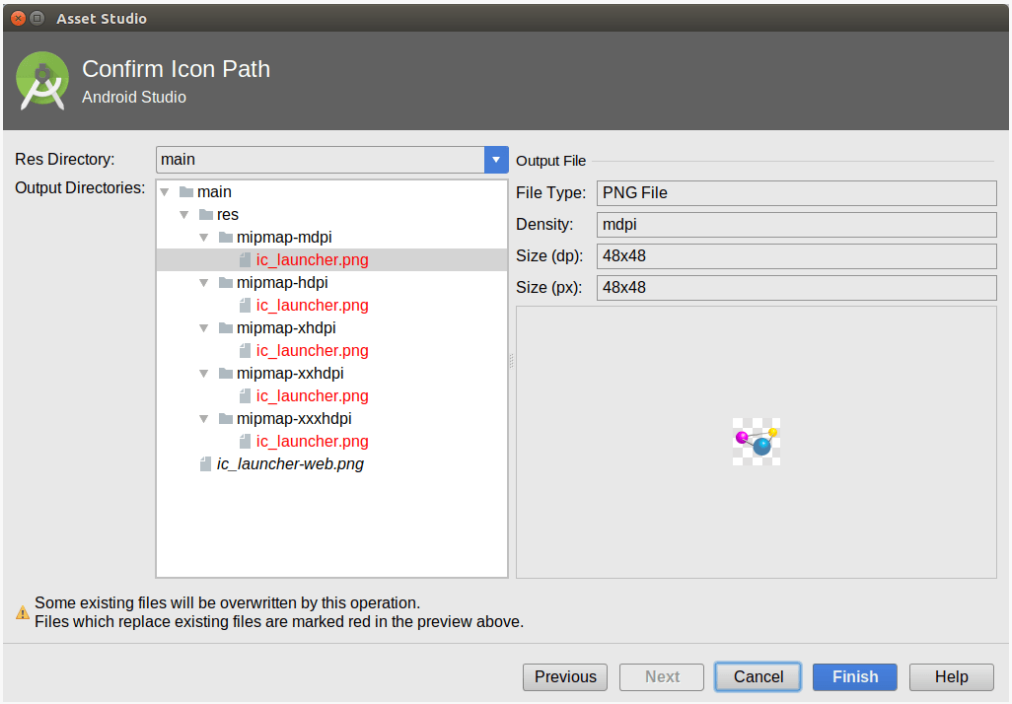


Figure 62: Asset Studio Wizard, Second Page

You should get a warning towards the bottom, indicating that if you finish the wizard, you will overwrite existing files. This is expected, as we are trying to replace the old `ic_launcher.png` files with new ones. So, go ahead and click Finish.

Step #3: Running the Result

If you run the resulting app, you will see that it shows up with the new name and icon, such as in the launcher:

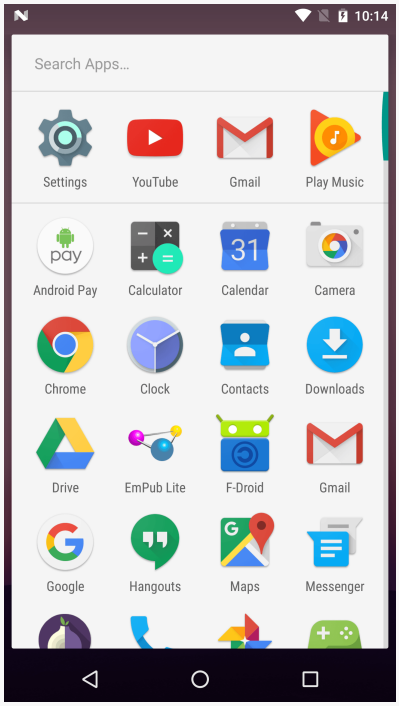


Figure 63: EmPubLite with New Icons

In Our Next Episode…

… we will add a progress indicator to the UI of our tutorial project.

The Theory of Widgets

There is a decent chance that you have already done work with widget-based UI frameworks. In that case, much of this chapter will be review, though checking out the section on the absolute positioning anti-pattern should certainly be worthwhile.

There is a chance, though, that your UI background has come from places where you have not been using a traditional widget framework, where either you have been doing all of the drawing yourself (e.g., game frameworks) or where the UI is defined more in the form of a document (e.g., classic Web development). This chapter is aimed at you, to give you some idea of what we are talking about when discussing the notion of widgets and containers.

What Are Widgets?

Wikipedia has a nice definition of a widget:

> In computer programming, a widget (or control) is an element of a graphical user interface (GUI) that displays an information arrangement changeable by the user, such as a window or a text box. The defining characteristic of a widget is to provide a single interaction point for the direct manipulation of a given kind of data. In other words, widgets are basic visual building blocks which, combined in an application, hold all the data processed by the application and the available interactions on this data.

(quote from the 7 March 2014 version of the page)

Take, for example, this Android screen:

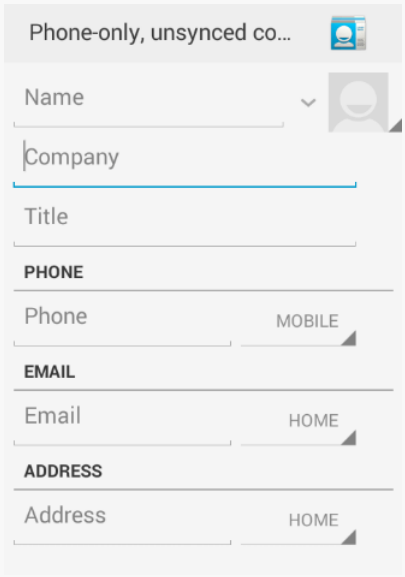


Figure 64: A Sample Android Screen

Here, we see:

- some text, like “Phone-only, unsynced co...” and “PHONE”
- an icon of a contact “Rolodex” card
- some data entry fields with hints like “Name” and “Company”
- some “spinner” drop-down lists (the items with the arrowheads pointing southeast)
- some gray divider lines

Everything listed above is a widget. The user interface for most Android screens (“activities”) is made up of one or more widgets.

This does not mean that you cannot do your own drawing. In fact, all the existing widgets are implemented via low-level drawing routines, which you can use for everything from your own custom widgets to games.

This also does not mean that you cannot use Web technologies. In fact, we will see later in this book a widget designed to allow you to embed Web content into an Android activity.

However, for most non-game applications, your Android user interface will be made up of several widgets.

Size, Margins, and Padding

Widgets have some sort of size, since a zero-pixel-high, zero-pixel-wide widget is not especially user-friendly. Sometimes, that size will be dictated by what is inside the widget itself, such as a label (`TextView`) having a size dictated by the text in the label. Sometimes, that size will be dictated by the size of whatever holds the widget (a "container", described in the next section), where the widget wants to take up all remaining width and/or height. Sometimes, that size will be a specific set of dimensions.

Widgets can have margins. As with CSS, margins provide separation between a widget and anything adjacent to it (e.g., other widgets, edges of the screen). Margins are really designed to help prevent widgets from running right up next to each other, so they are visually distinct. Some developers, however, try to use margins as a way to hack "absolute positioning" into Android, which is an anti-pattern that we will examine later in this chapter.

Widgets can have padding. As with CSS, padding provides separation between the contents of a widget and the widget's edges. This is mostly used with widgets that have some sort of background, like a button, so that the contents of the widget (e.g., button caption) does not run right into the edges of the button, once again for visual distinction.

What Are Containers?

Containers are ways of organizing multiple widgets into some sort of structure. Widgets do not naturally line themselves up in some specific pattern — we have to define that pattern ourselves.

In most GUI toolkits, a container is deemed to have a set of children. Those children are widgets, or sometimes other containers. Each container has its basic rule for how it lays out its children on the screen, possibly customized by requests from the children themselves.

Common container patterns include:

- put all children in a row, one after the next
- put all children in a column, one below the next
- arrange the children into a table or grid with some number of rows and columns

- anchor the children to the sides of the container, according to requests made by those children
- anchor the children to other children in the container, according to requests made by those children
- stack all children, one on top of the next
- and so on

In the sample activity above, the dominant pattern is a column, with things laid out from top to bottom. Some of those things are rows, with contents laid out left to right. However, as it turns out, the area with most of those widgets is scrollable.

Android supplies a handful of containers, designed to handle most common scenarios, including everything in the list above. You are also welcome to create your own custom containers, to implement business rules that are not directly supported by the existing containers.

Note that containers also have size, padding, and margins, just as widgets do.

The Absolute Positioning Anti-Pattern

You might wonder why all of these containers and such are necessary. After all, can't you just say that such-and-so widget goes at this pixel coordinate, and this other widget goes at that pixel coordinate, and so on?

Many developers have taken that approach — known as absolute positioning – over the years, to their eventual regret.

For example, many of you may have used Windows apps, back in the 1990's, where when you would resize the application window, the app would not really react all that much. You would expand the window, and the UI would not change, except to have big empty areas to the right and bottom of the window. This is because the developers simply said that such-and-so widget goes at this pixel coordinate, and this other widget goes at that pixel coordinate, **regardless of the actual window size**.

In modern Web development, you see this in the debate over fixed versus fluid Web design. The consensus seems to be that fluid designs are better, though frequently they are more difficult to set up. Fluid Web designs can better handle differing browser window sizes, whether those window sizes are because the user resized their browser window manually, or because those window sizes are dictated by the

screen resolution of the device viewing the Web page. Fixed Web designs — effectively saying that such-and-so element goes at such-and-so pixel coordinate and so on — tend to be easier to build but adapt more poorly to differing browser window sizes.

In mobile, particularly with Android, we have a wide range of possible screen resolutions, from QVGA (320x240) to beyond 1080p (1920x1080), and many values in between. Moreover, any device manufacturer is welcome to create a device with whatever resolution they so desire – there are no rules limiting manufacturers to certain resolutions. Hence, as developers, having the Android equivalent of fluid Web designs is critical, and the way you will accomplish that is by sensible use of containers, avoiding absolute positioning. The containers (and, to a lesser extent, the widgets) will determine how extra space is employed, as the screens get larger and larger.

The Theme of This Section: Themes

In Web development, we have had stylesheets for quite a while. Through such Cascading Style Sheets (CSS) files, we can stipulate various rules about how our Web pages should look. This includes:

- Establishing a default look for certain HTML tags by tag name (e.g., setting the font and size for all `<h1>` and `<h2>` elements)
- Establishing a look for specific HTML elements by class or ID (e.g., setting the width of a specific `<div>` to a certain number of CSS pixels)

In Android, the equivalent concepts can be found in styles and themes. Styles are a collection of values for properties (e.g., have a foreground color of red). These can be applied to specific widgets (e.g., this label should adopt this style), or they can be employed by "themes" that provide the default look for all sorts of widgets and other elements of our UI.

Of course, you do not have to declare any theme for your app. Android will give you a default look-and-feel without any specific theme. That look-and-feel has varied over the years, though, affecting the visual fundamentals of various Android widgets. These themes have names by which we refer to them: `Theme`, `Theme.Holo`, and `Theme.Material`.

In the Beginning, There Was "Theme", And It Was Meh

Way back in Android 1.0, the default theme was known simply as `Theme`. Technically, all themes inherit from `Theme`, much as how later CSS stylesheets effectively "inherit" the settings established by prior stylesheets.

The `Theme` UI had a particular look to it:

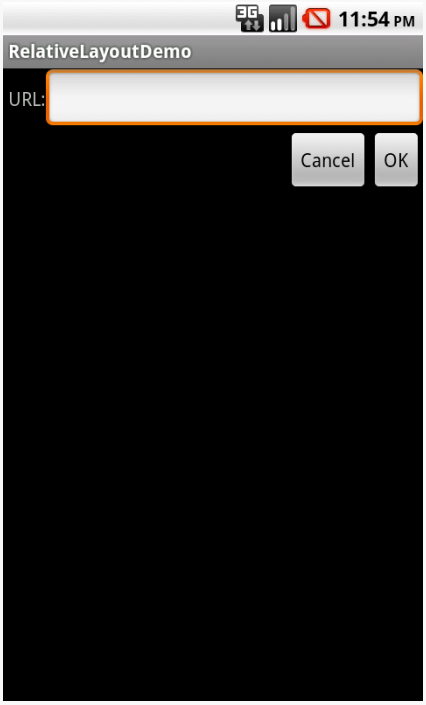


Figure 65: Labels, Fields, and Buttons in Theme

For example:

- At the top of the screen, we had a thin gray "title bar" with the name of our app
- The focused field (an `EditText` widget) had a bright orange outline, whereas normally it was a plain white rectangle
- The buttons ("OK" and "Cancel") were... well... buttons

Holo, There!

Android 3.0 (API Level 11) introduced a new default theme, `Theme.Holo`, with the so-called "holographic widget theme". This changed the look of our UI somewhat:

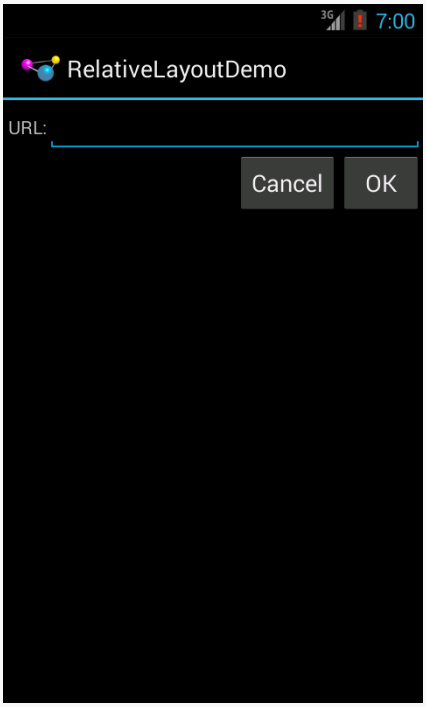


Figure 66: Labels, Fields, and Buttons in Theme.Holo

Now:

- At the top of the screen, we have an "action bar", containing our app's logo and name
- The focused field has a blue "underbracket", whereas normally it is gray
- The buttons are styled slightly differently, with a bigger font, alternative backgrounds, etc.

Considering the Material

Android 5.0 changed the default theme yet again, to `Theme.Material`:

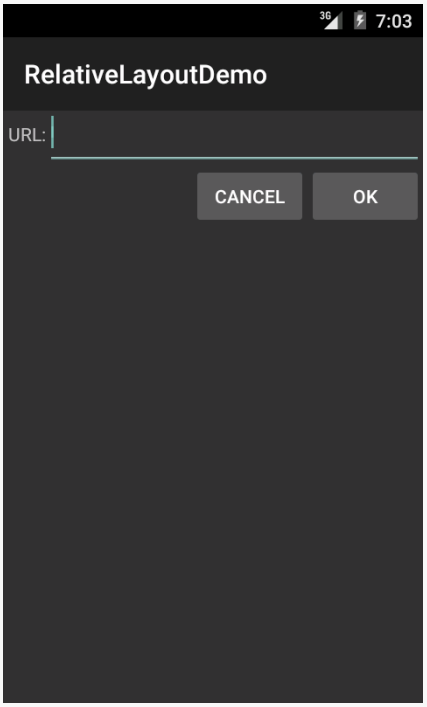


Figure 67: Labels, Fields, and Buttons in Theme.Material

Now:

- The action bar at the top of the screen no longer shows the app icon
- Our field is indicated by an underline, which is teal when focused or gray when unfocused
- The buttons are now forced into all-caps font, with a slightly smaller font size and subtly different background than we had with `Theme.Holo`

Doing More with Themes

Of course, we can do a lot more than just use these. There are other stock themes, with different characteristics. Furthermore, we can customize the themes, by defining our own (inheriting from a stock theme) and changing some of the properties (e.g., replacing the teal color with something else).

We will get much more into creating custom styles and themes later in the book.

However, we will see the effects of `Theme`, `Theme.Holo`, and `Theme.Material` on stock widgets in an upcoming chapter.

The Android User Interface

The project you created in an earlier tutorial was just the default files generated by the Android build tools — you did not write any Java code yourself. In this chapter, we will examine the basic Java code and resources that make up an Android activity.

The Activity

The Java source code that you maintain will be in a standard Java-style tree of directories based upon the Java package you chose when you created the project (e.g., `com.commonsware.android` results in `com/commonsware/android/`). Android Studio will have that source, by default, in `app/src/main/java/` off of the top-level project root.

If, in the new-project wizard, you elected to create an activity, you will have, in the innermost directory, a Java source file representing an activity class.

A very simple activity looks like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;

public class EmPubLiteActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
}
```

(from EmPubLite-AndroidStudio/T2-Project/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Dissecting the Activity

Let's examine this Java code piece by piece:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;
```

(from EmPubLite-AndroidStudio/T2-Project/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

By default, the package declaration is the same as the one you used when creating the project. And, like any other Java project, you need to import any classes you reference. Most of the Android-specific classes are in the `android` package.

Remember that not every Java SE class is available to Android programs! Visit the Android class reference to see what is and is not available.

```
public class EmPubLiteActivity extends Activity {
```

(from EmPubLite-AndroidStudio/T2-Project/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Activities are public classes, inheriting from the `android.app.Activity` base class (or, possibly, from some other class that itself inherits from `Activity`). You can have whatever data members you decide that you need, though the initial code has none.

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```

(from EmPubLite-AndroidStudio/T2-Project/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

The `onCreate()` method is invoked when the activity is started. We will discuss the `Bundle` parameter to `onCreate()` in a later chapter. For the moment, consider it an opaque handle that all activities receive upon creation.

The first thing you normally should do in `onCreate()` is chain upward to the superclass, so the stock Android activity initialization can be done. The only other statement in our stub project's `onCreate()` is a call to `setContentView()`. This is where we tell Android what the user interface is supposed to be for our activity.

This raises the question: what does `R.layout.main` mean? Where did this `R` come from?

To explain that, we need to start thinking about layout resources and how resources are referenced from within Java code.

Using XML-Based Layouts

As noted in the previous chapter, Android uses a series of widgets and containers to describe your typical user interface. These all inherit from an `android.view.View` base class, for things that can be rendered into a standard widget-based activity.

While it is technically possible to create and attach widgets and containers to our activity purely through Java code, the more common approach is to use an XML-based layout file. Dynamic instantiation of widgets is reserved for more complicated scenarios, where the widgets are not known at compile-time (e.g., populating a column of radio buttons based on data retrieved off the Internet).

With that in mind, it's time to break out the XML and learn how to lay out Android activity contents that way.

What Is an XML-Based Layout?

As the name suggests, an XML-based layout is a specification of its widgets' relationships to each other — and to containers — encoded in XML format. Specifically, Android considers XML-based layouts to be resources, and as such layout files are stored in the `res/layout/` directory inside your Android project (or, as we will see later, other layout resource sets, like `res/layout-land/` for layouts to use when the device is held in landscape). As has been noted elsewhere in this book, the initial location of `res/` is in `app/src/main/` for Android Studio.

Each XML file contains a tree of elements specifying a layout of widgets and containers that make up one `View`. The attributes of the XML elements are properties, describing how a widget should look or how a container should behave. For example, if a `Button` element has an attribute value of `android:textStyle = "bold"`, that means that the text appearing on the face of the button should be rendered in a boldface font style.

For example, here is a `res/layout/main.xml` file that could be used with the aforementioned activity:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout android:id="@+id/main"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingBottom="@dimen/activity_vertical_margin"
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight="@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
  tools:context="com.commonsware.empublite.EmPubLiteActivity">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />
</RelativeLayout>
```

(from EmPubLite-AndroidStudio/T2-Project/EmPubLite/app/src/main/res/layout/main.xml)

The class name of a widget or container — such as `RelativeLayout` or `TextView` – forms the name of the XML element. Since `TextView` is an Android-supplied widget, we can just use the bare class name. If you create your own widgets as subclasses of `android.view.View`, you would need to provide a full package declaration as well (e.g., `com.commonsware.android.MyWidget`).

The root element needs to declare the Android XML namespace (`xmlns:android="http://schemas.android.com/apk/res/android"`). All other elements will be children of the root and will inherit that namespace declaration.

The attributes are properties of the widget or container, describing what it should look and work like. For example, the `android:layout_width` and `android:layout_height` attributes on the `TextView` element make a request from the child (`TextView`) to its parent (a `RelativeLayout`) for what size the child should be.

We will get into details about these attributes, their possible values, and their uses, in upcoming chapters. Note that those attributes in the `tools` namespace (e.g., `tools:context`) are there solely to support the Android build tools, and do not affect the runtime execution of your project.

Android's SDK ships with a tool (`aapt`) which uses the layouts. This tool will be automatically invoked by your Android tool chain (e.g., Android Studio). Of particular importance to you as a developer is that `aapt` generates an `R.java` source

file, allowing you to access layouts and widgets within those layouts directly from your Java code. In other words, this is where that magic `R` value used in `setContentView()` comes from. We will discuss that a bit more later in this chapter.

XML Layouts and Your IDE

If you are using Android Studio, and you double-click on the `res/layout/main.xml` file in your project, you will not initially see that XML. Instead, you will be taken to the graphical layout editor:

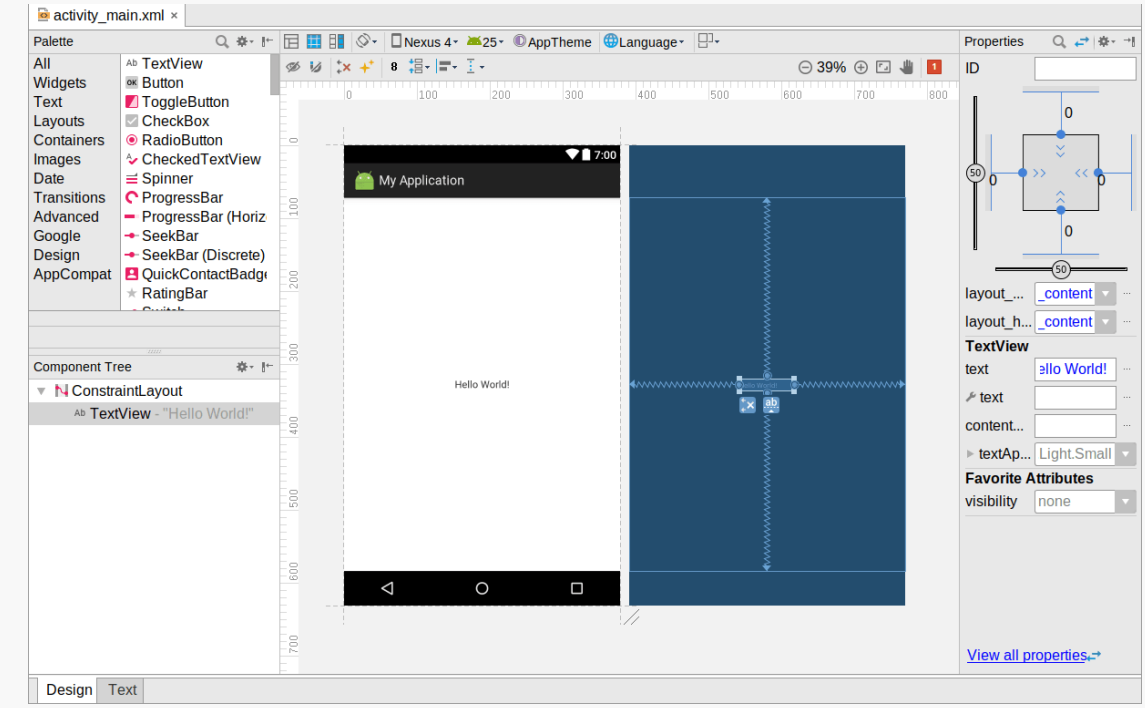


Figure 68: Android Studio Graphical Layout Editor

We will go into much more detail about using the graphical layout editor in an upcoming chapter, as we start to work more with specific widgets and containers.

Why Use XML-Based Layouts?

Almost everything you do using XML layout files can be achieved through Java code. For example, you could use `setText()` to have a button display a certain caption, instead of using a property in an XML layout. Since XML layouts are yet another file for you to keep track of, we need good reasons for using such files.

Perhaps the biggest reason is to assist in the creation of tools for view definition, such as the aforementioned graphical layout editors in Android Studio. Such GUI builders could, in principle, generate Java code instead of XML. The challenge is re-reading the definition in to support edits — that is far simpler if the data is in a structured format like XML than in a programming language. Moreover, keeping the generated bits separated out from hand-written code makes it less likely that somebody's custom-crafted source will get clobbered by accident when the generated bits get re-generated. XML forms a nice middle ground between something that is easy for tool-writers to use and easy for programmers to work with by hand as needed.

Also, XML as a GUI definition format is becoming more commonplace. Microsoft's XAML, Adobe's Flex, Google's GWT, and Mozilla's XUL all take a similar approach to that of Android: put layout details in an XML file and put programming smarts in source files (e.g., JavaScript for XUL). Many less-well-known GUI frameworks, such as ZK, also use XML for view definition. While "following the herd" is not necessarily the best policy, it does have the advantage of helping to ease the transition into Android from any other XML-centered view description language.

Using Layouts from Java

Given that you have painstakingly set up the widgets and containers for your view in an XML layout file named `main.xml` stored in `res/layout/`, all you need is one statement in your activity's `onCreate()` callback to use that layout, as we saw in our stub project's activity:

```
setContentView(R.layout.main);
```

Here, `R.layout.main` tells Android to load in the layout (`layout`) resource (`R`) named main.xml (`main`).

Basic Widgets

Every GUI toolkit has some basic widgets: fields, labels, buttons, etc. Android's toolkit is no different in scope, and the basic widgets will provide a good introduction as to how widgets work in Android activities. We will examine a number of these in this chapter.

Common Concepts

There are a few core features of widgets that we need to discuss at the outset, before we dive into details on specific types of widgets.

Widgets and Attributes

As mentioned in a previous chapter, widgets have attributes that describe how they should behave. In an XML layout file, these are literally XML attributes on the widget's element in the file. Usually, there are corresponding getter and setter methods for manipulating this attribute at runtime from your Java code.

If you visit the JavaDocs for a widget, such as the JavaDocs for `TextView`, you will see an "XML Attributes" table near the top. This lists all of the attributes defined uniquely on this class, and the "Inherited XML Attributes" table that follows lists all those that the widget inherits from superclasses, such as `View`. Of course, the JavaDocs also list the fields, constants, constructors, and public/protected methods that you can use on the widget itself.

This book does not attempt to explain each and every attribute on each and every widget. We will, however, cover the most popular widgets and the most commonly-used attributes on those widgets.

Referencing Widgets By ID

Many widgets and containers only need to appear in the XML layout file and do not need to be referenced in your Java code. For example, a static label (`TextView`) frequently only needs to be in the layout file to indicate where it should appear.

Anything you *do* want to use in your Java source, though, needs an `android:id`.

The convention is to use `@+id/...` as the `id` value (where the `...` represents your locally-unique name for the widget) for the first occurrence of a given `id` value in your layout file. The second and subsequent occurrences in the same layout file should drop the `+` sign.

Android provides a few special `android:id` values, of the form `@android:id/...` — we will see some of these in various chapters of this book.

To access our identified widgets, use `findViewById()`, passing it the numeric identifier of the widget in question. That numeric identifier was generated by Android in the `R` class as `R.id.something` (where `something` is the specific widget you are seeking).

This concept will become important as we try to attach listeners to our widgets (e.g., finding out when a checkbox is checked) or when we try referencing widgets from other widgets in a layout XML file (e.g., with `RelativeLayout`). All of this will be covered later in this chapter.

The Curious Case of the Cast

Most sample code that you will see for Android will show the results of the `findViewById()` call being cast to some other class:

```
TextView tv=(TextView)findViewById(R.id.name);
```

That is because for most of Android's existence, `findViewById()` returned a `View`, so we would need to down-cast that to a more appropriate class.

However, if you are using Android Studio 3.0+, with an app with a `compileSdkVersion` of 26 or higher, you will notice that you no longer need those casts. Any existing ones will show up in gray, with a tooltip indicating that the cast is unnecessary.

What happened is that Android 8.0 changed `findViewById()` to return `T`, using Java generics to automatically cast it to the data type you request in the assignment. Casts are a compile-time thing in Java — they do not appear in compiled code and have no effects at runtime. As a result, code that skips the casts works perfectly fine on older devices as well.

Size

Most of the time, we need to tell Android how big we want our widgets to be. Occasionally, this will be handled for us — we will see an example of that with `TableLayout` in an upcoming chapter. But generally we need to provide this information ourselves.

To do that, you need to supply `android:layout_width` and `android:layout_height` attributes on your widgets in the XML layout file. These attributes' values have three flavors:

1. You can provide a specific dimension, such as `125dip` to indicate the widget should take up exactly a certain size (here, 125 density-independent pixels)
2. You can provide `wrap_content`, which means the widget should take up as much room as its contents require (e.g., a `TextView` label widget's content is the text to be displayed)
3. You can provide `match_parent`, which means the widget should fill up all remaining available space in its enclosing container

The latter two flavors are the most common, as they are independent of screen size, allowing Android to adjust your view to fit the available space.

Note that you will also see `fill_parent`. This is an older synonym for `match_parent`. `match_parent` is the recommended value going forward, but `fill_parent` will certainly work.

This chapter focuses on individual widgets. Size becomes much more important when we start combining multiple widgets on the screen at once, and so we will be spending more time on sizing scenarios in later chapters.

The `layout_` prefix on these attributes means that these attributes represent requests by the widget to its enclosing container. Whether those requests will be truly honored will depend a bit on what other widgets there are in the container and what *their* requests are.

Introducing the Graphical Layout Editor

If you open a layout resource in Android Studio, by default you will see the graphical layout editor:

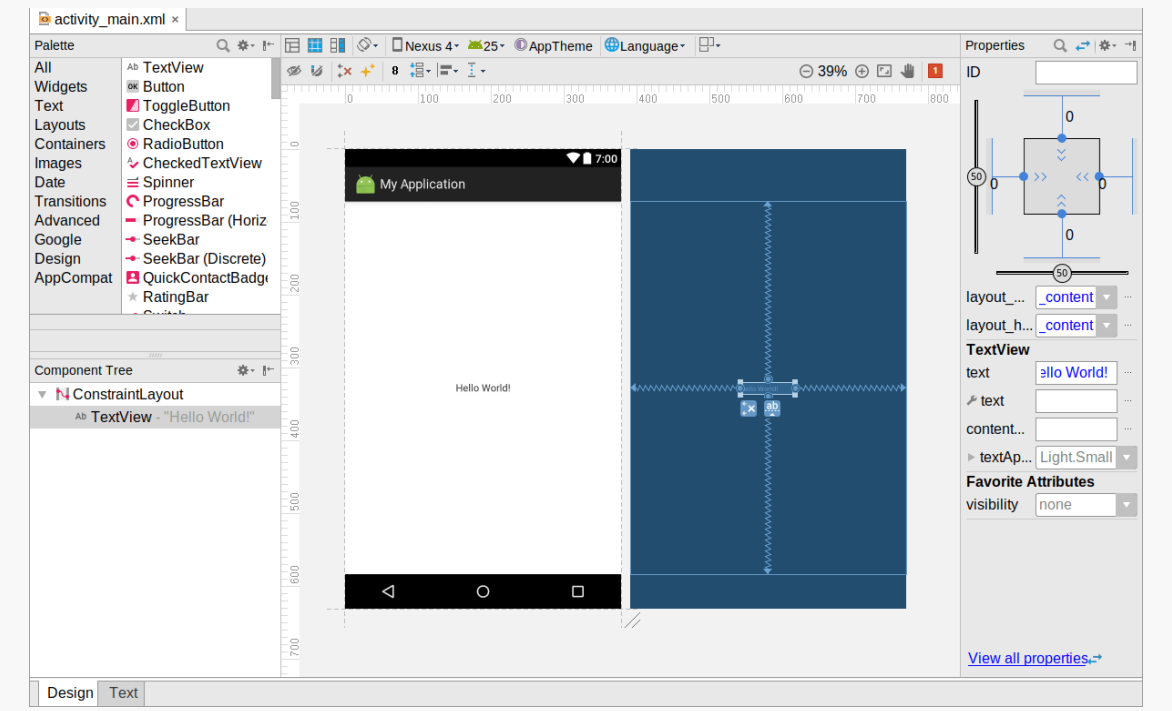


Figure 69: Android Studio Graphical Layout Editor

This offers a drag-and-drop means of defining the contents of that layout resource.

Android IDEs have had drag-and-drop GUI building capability for several years, dating back to when Eclipse was the official IDE. However, Android Studio 2.2 made some significant changes in the way the drag-and-drop GUI builder looks and works, and later updates have changed it further. This book covers the current look-and-feel, but older blog posts, Stack Overflow answers, and similar resources may refer to aspects of previous GUI builders.

With all that in mind, let's look at the different pieces of the graphical layout editor.

Palette

The upper-left side of the graphical layout editor is the Palette tool:

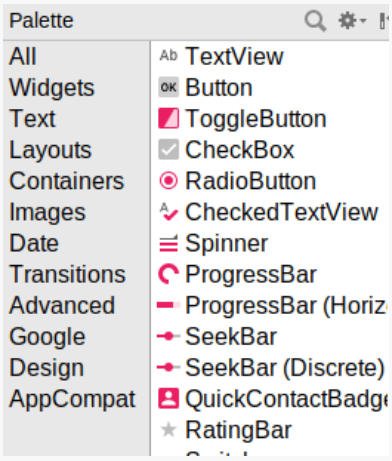


Figure 70: Palette Tool

This lists all sorts of widgets and containers that you can drag and drop. They are divided into categories ("Widgets", "Text", "Layouts", etc.) with many options in each. A few are not strictly widgets or containers but rather other sorts of XML elements that you can have in a layout resource (e.g., `<fragment>`, `<requestFocus>`).

As we cover how to use the graphical layout editor, we will see how to create and configure several of these widgets, containers, and other items.

Preview

The main central area of the graphical layout editor consists of two perspectives on your layout resource contents. The one on the left is a preview of what your UI should resemble, if this layout were used for the UI of an activity:

Figure 71: UI Preview

This pours your layout resource contents into a preview frame that has aspects of a regular Android device, such as the navigation bar at the bottom and the status bar at the top.

If you drag items out of the Palette and drop them into the preview area, they will be added to your layout resource.

Blueprint

To the right of the preview area is the blueprint view. This also visually depicts your layout resource. However, rather than showing you a preview of what your UI might look like, it visually represents what widget and container classes you are using. And, for some types of containers, it will show some of the sizing and positioning rules that you are using for children of that container:

Figure 72: Blueprint

For a trivial layout resource, the blueprint view does not show you much. It will become more useful with more complex layout resources. In particular, it is very useful when you have designated some widgets or containers as being invisible, as they will show up in the blueprint but not in the preview:

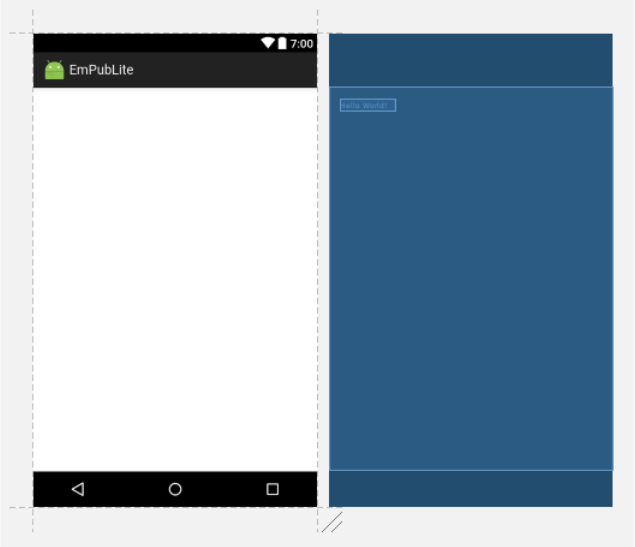


Figure 73: Layout Resource with Invisible TextView

Preview Toolbar

Above the preview and blueprint is a bi-level toolbar that allows you to configure various aspects of the preview and blueprint appearance and behavior.

Upper Toolbar Level

From left to right, the upper level of the toolbar contains:

- A drop-down to toggle whether you see the preview, the blueprint, or both
- A toggle to control whether you are seeing the layout as applied to portrait or landscape perspectives
- A drop-down to choose what device size and resolution should be used for the preview, culled from your emulator images and the available device definitions
- A drop-down to choose what API level should be used for the simulated UI of the preview
- A button to choose what theme to use for presenting the UI of the preview
- A button to choose what language to use for determining which of your string resources gets used in the preview

Figure 74: Preview Toolbar, Top Level

A couple of those — particularly the theme selector — pertain to topics that we will explore later in the book.

Lower Toolbar Level

On the right side of the lower toolbar level are:

- Zoom controls
- A button to reset the zoom to fill the area available for the preview and/or blueprint
- A button to enable “Pan and Zoom” mode (more on this shortly)
- An “info” icon indicating if there are any warnings or errors associated with your layout resource

Figure 75: Preview Toolbar, Bottom Level, Right Side

The left side of the bottom level of the preview toolbar will change, based upon the selected widget or container, offering options for you to be able to make simple changes to whatever is selected. We will see examples of this over the next few chapters

Pan and Zoom Mode

The "Pan and Zoom" mode allows you to navigate a zoomed-in version of your preview and/or blueprint. While the main views will be zoomed in, tapping the "Pan and Zoom" toolbar button opens up a floating window showing the entire preview and/or blueprint, with a red box indicating what portion of that you are seeing in the main views:

Figure 76: Preview and Blueprint, with Pan and Zoom Enabled

You can drag the red frame around the floating window to change what portion of the entire zoomed-in view you can see in the main view.

Component Tree

Towards the bottom-left corner is the component tree:

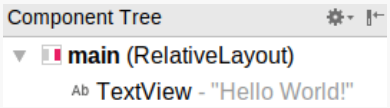


Figure 77: Component Tree

This gives you a full tree of all of the widgets and containers inside of this layout resource. It corresponds to the tree of XML elements in the layout resource itself.

Clicking on any item in the component tree highlights it in both the preview and blueprint views, plus it switches to that widget or container for the attributes pane.

Attributes

When a widget or container is selected — whether via the component tree, clicking on it in the preview, or clicking on it in the blueprint – the Attributes pane on the right will allow you to manipulate how that widget or container looks and behaves.

By default, it will bring up a condensed roster of the most important attributes:

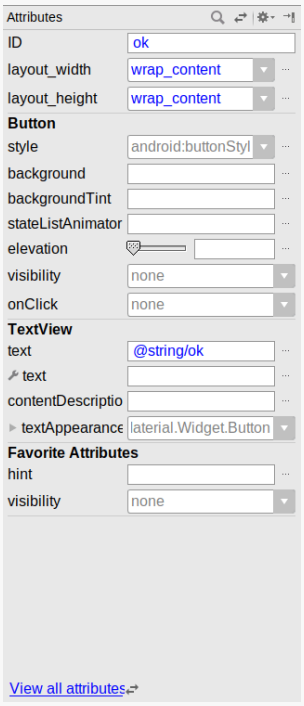


Figure 78: Attributes Pane, Showing Condensed Roster

Clicking the “View all attributes” link, or the opposing-arrows toolbar button, switches to a list of all attributes:

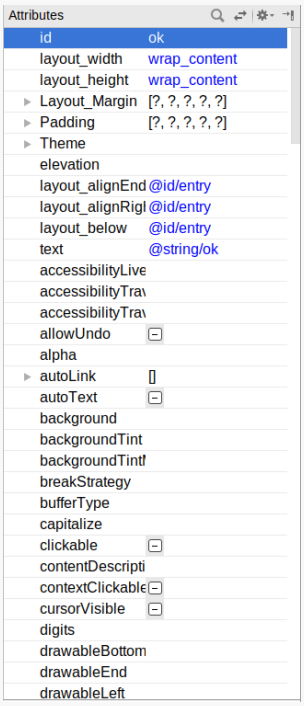


Figure 79: Attributes Pane, Showing Full Roster

You can also click the magnifying glass icon in the toolbar of this pane to search for available attributes by name:

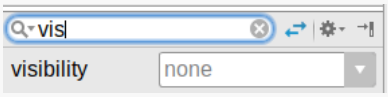


Figure 80: Attributes Pane, Showing Search Results

We will see what many of these attributes are and how to work with them over the course of the next few chapters.

For the attributes in the full roster, you can click the star icon on the left to mark them as “favorites”:

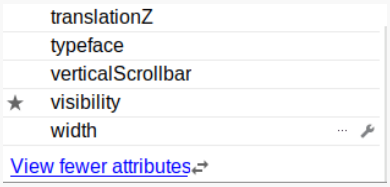


Figure 81: Attributes Pane, Showing One Favorite Attribute

Those favorite attributes show up in the condensed roster, in the section labeled "Favorite Attributes".

Text Tab

Towards the bottom of the graphical layout tool, you will see that it contains two sub-tabs. One, "Design", encompasses everything described above. The other, "Text", allows you to edit the raw XML that is the actual content of the layout resource:

Figure 82: Text Sub-Tab in Layout Editor

By default, the entire area is devoted to the text editor. However, when the Text sub-tab is active, a "Preview" tool will appear docked on the right side of the Android Studio window. Clicking that will display the preview from the Design sub-tab:

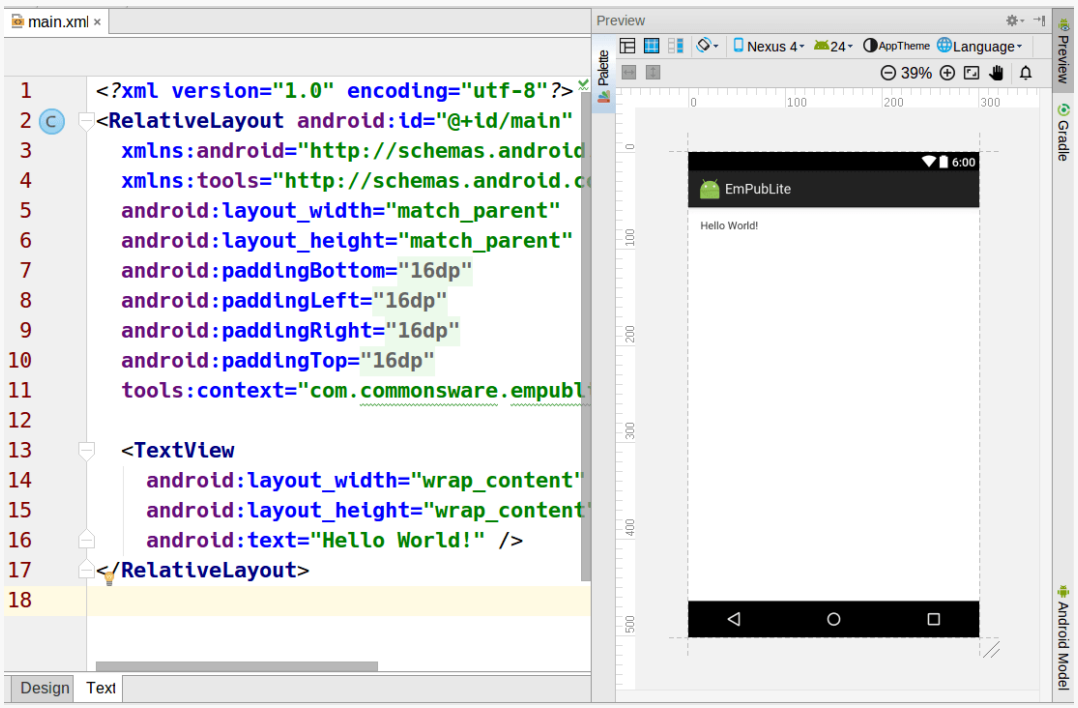


Figure 83: Text Sub-Tab with Preview

Clicking on items in the preview will highlight the corresponding XML element in the text editor.

And Now, Some Notes About the Book's Sample Projects

This book profiles hundreds of sample apps, demonstrating everything from how to display text on the screen to how to scan NFC tags or work with the clipboard.

These sample apps are housed in a Git repository on GitHub. They are all open source, and the *vast* majority are licensed under the Apache Software License 2.0, in case you copy any of the code.

The Git repo uses tags that match the book version numbers (e.g., the `v8.8` tag is for Version 8.8 of the book). Hence, to get the source code that matches your book version, you can:

- Clone the repo (e.g., `git clone https://github.com/commonsguy/cw-omnibus.git`) and then checkout a particular tag; or

- Download the ZIP archive associated with a particular book version

For those projects that you wish to examine in Android Studio, you can use File > New > Import Project to import the project. However, you have to do this on a project-by-project basis, importing each project directory as needed. Roughly speaking, any directory that has `settings.gradle` in it is a project directory, particularly with respect to this book's samples.

Starting with the next section, the book will be displaying code from these sample projects. Links will point you to a particular sample project or to a specific file in that project, such as a layout resource or Java source file. Those links go to the GitHub repository, but you can use that information to help you identify where the corresponding file is on your development machine, if you elected to download the code.

Assigning Labels

The simplest widget is the label, referred to in Android as a `TextView`. Like in most GUI toolkits, labels are bits of text not editable directly by users. Typically, they are used to identify adjacent widgets (e.g., a "Name:" label before a field where one fills in a name).

In Java, you can create a label by creating a `TextView` instance. More commonly, though, you will create labels in XML layout files by adding a `TextView` element to the layout, with an `android:text` attribute to set the value of the label itself. If you need to swap labels based on certain criteria, such as internationalization, you may wish to use a string resource reference in `android:text` instead (e.g., `@string/label`).

For example, in our last tutorial, we still are using the automatically-generated `res/layout/main.xml` file, containing, among other things, a `TextView`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout android:id="@+id/main"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingBottom="@dimen/activity_vertical_margin"
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight="@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
```

```
  tools:context="com.commonsware.empublite.EmPubLiteActivity">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />

</RelativeLayout>
```

(from EmPubLite-AndroidStudio/T4-Resources/EmPubLite/app/src/main/res/layout/main.xml)

Android Studio Graphical Layout Editor

The `TextView` widget is available in the "Widgets" category of the Palette in the Android Studio graphical layout editor:

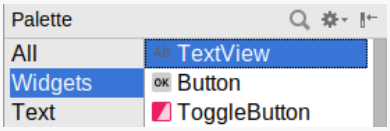


Figure 84: Palette, "Plain TextView" in Widgets Category

You can drag that `TextView` from the palette into a layout file in the main editing area to add the widget to the layout. Or, drag it over the top of some container you see in the Component Tree pane of the editor to add it as a child of that specific container.

Clicking on the new `TextView` will set up the Attributes pane with the various attributes of the widget, ready for you to change as needed.

Editing the Text

The "Text" attribute will allow you to choose or define a string resource to serve as the text to be displayed:

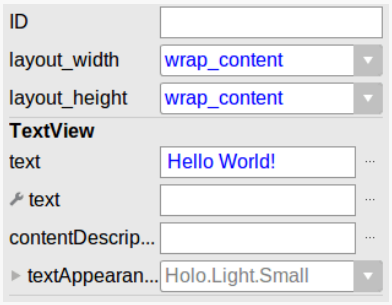


Figure 85: Attributes Pane, Showing TextView "text" Attribute

The "text" with a wrench icon allows you to provide a separate piece of text that will show up in the preview, but not be used by your app at runtime.

You can either type a literal string right in the Attributes pane row, or you can click the "..." button to the right of the field to pick a string resource:

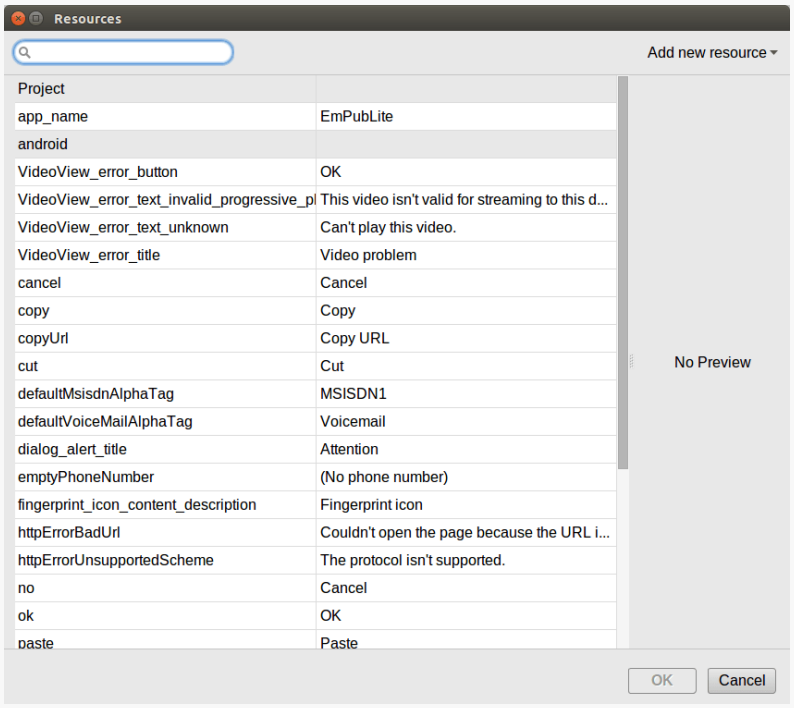


Figure 86: String Resources Dialog

You can highlight one of those resources and click "OK" to use it. Or, towards the upper-right of that dialog, there is an "Add new resource" drop-down. When viewing string resources, that drop-down will contain a single command: "New string

Value...". Choosing it will allow you to define a new string resource via another dialog:

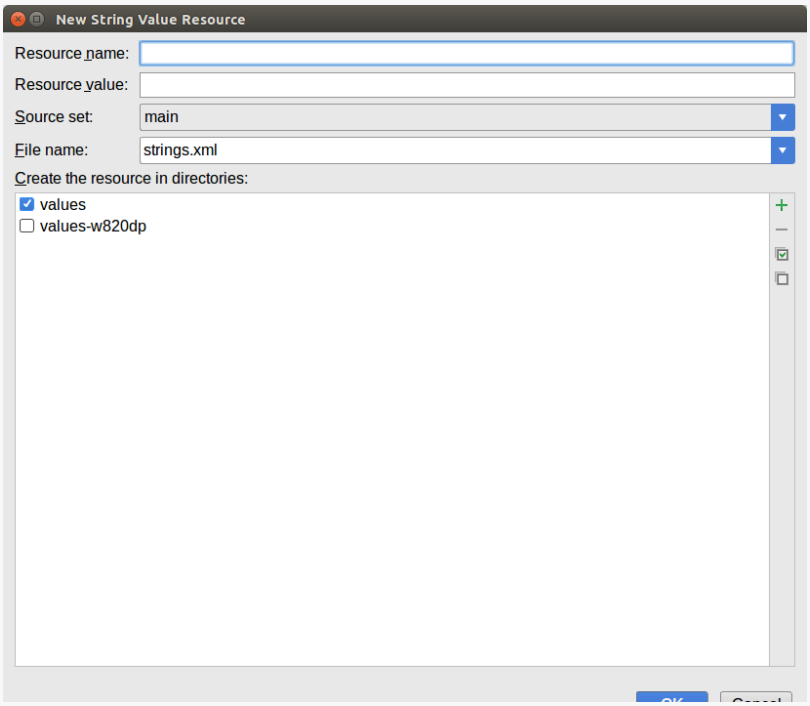


Figure 87: New String Resource Dialog

You can give your new string resource a name, the actual text of the string itself, the filename in which the string resource should reside (`strings.xml` by default), and which `values/` directory the string should go into (`values` by default). You will also choose the "source set" — for now, that will just be `main`. Once you accept the dialog, your new string resource will be applied to your `TextView`.

Editing the ID

The "id" attribute will allow you to change the `android:id` value of the widget:

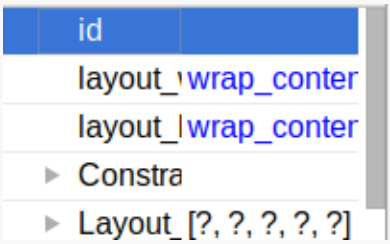


Figure 88: Attributes Pane, with TextView "id" Attribute Selected

The value you fill in here is what goes after the `@+id/` portion (e.g., `textView2`).

Notable TextView Attributes

`TextView` has numerous other attributes of relevance for labels, such as:

1. `android:typeface` to set the typeface to use for the label (e.g., `monospace`)
2. `android:textStyle` to indicate that the typeface should be made bold (`bold`), italic (`italic`), or bold and italic (`bold_italic`)
3. `android:textColor` to set the color of the label's text, in RGB hex format (e.g., `#FF0000` for red) or ARGB hex format (e.g., `#88FF0000` for a translucent red)

These attributes, like most others, can be modified through the Attributes pane.

For example, in the `Basic/Label` sample project, you will find the following layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@string/profound"
  />
```

(from Basic/Label/app/src/main/res/layout/main.xml)

Just that layout alone, with the stub Java source provided to your app, along with appropriate string resources, gives you:

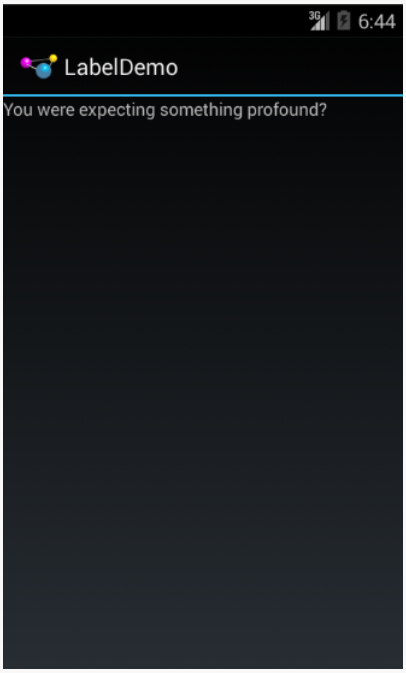


Figure 89: The LabelDemo Sample Application

A Commanding Button

Android has a `Button` widget, which is your classic push-button "click me and something cool will happen" widget. As it turns out, `Button` is a subclass of `TextView`, so everything discussed in the preceding section in terms of formatting the face of the button still holds.

For example, in the `Basic/Button` sample project, you will find the following layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button"/>
```

```
</LinearLayout>
```

(from Basic/Button/app/src/main/res/layout/main.xml)

Just that layout alone, with the stub Java source provided to your app, along with appropriate string resources, gives you:

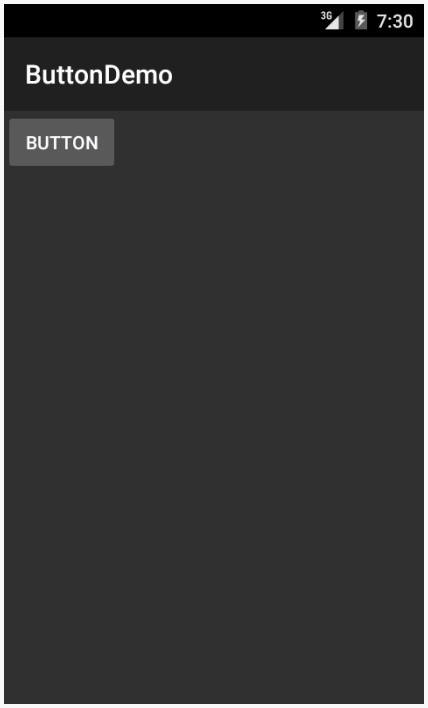


Figure 90: Button Widget

Android Studio Graphical Layout Editor

As with the `TextView` widget, the `Button` widget is available in the "Widgets" portion of the Palette in the Android Studio graphical layout editor:

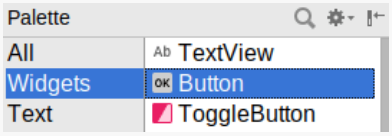


Figure 91: Widgets Palette, Button Shown Highlighted

You can drag that `Button` from the palette into a layout file in the main editing area to add the widget to the layout. The Attributes pane will then let you adjust the

various attributes of this `Button`. Since `Button` inherits from `TextView`, most of the options are the same (e.g., "Text").

Tracking Button Clicks

Buttons are command widgets — when the user presses a button, they expect something to happen.

To define what happens when you click a `Button`, you can:

1. Define some method on your `Activity` that holds the button that takes a single `View` parameter, has a `void` return value, and is `public`
2. In your layout XML, on the `Button` element, include the `android:onClick` attribute with the name of the method you defined in the previous step

For example, we might have a method on our `Activity` that looks like:

```
public void someMethod(View theButton) {
  // do something useful here
}
```

Then, we could use this XML declaration for the `Button` itself, including `android:onClick`:

```
<Button
  android:onClick="someMethod"
  ...
/>
```

This is enough for Android to "wire together" the `Button` with the click handler. When the user clicks the button, `someMethod()` will be called.

Another approach is to skip `android:onClick`, instead calling `setOnClickListener()` on the `Button` object in Java code. When a `Button` is used directly by an activity, this is not typically used — `android:onClick` is a bit cleaner. However, when we start to talk about fragments, you will see that `android:onClick` does not work that well with fragments, and so we will use `setOnClickListener()` at that point.

Fleeting Images

Android has two widgets to help you embed images in your activities: `ImageView` and `ImageButton`. As the names suggest, they are image-based analogues to `TextView` and `Button`, respectively.

Each widget takes an `android:src` attribute (in an XML layout) to specify what picture to use. These usually reference a drawable resource (e.g., `@drawable/icon`).

`ImageButton`, a subclass of `ImageView`, mixes in the standard `Button` behaviors, for responding to clicks and whatnot.

For example, take a peek at the `main.xml` layout from the `Basic/ImageView` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/icon"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:adjustViewBounds="true"
  android:src="@drawable/molecule"/>
```

(from Basic/ImageView/app/src/main/res/layout/main.xml)

The result, just using the code-generated activity, is simply the image:

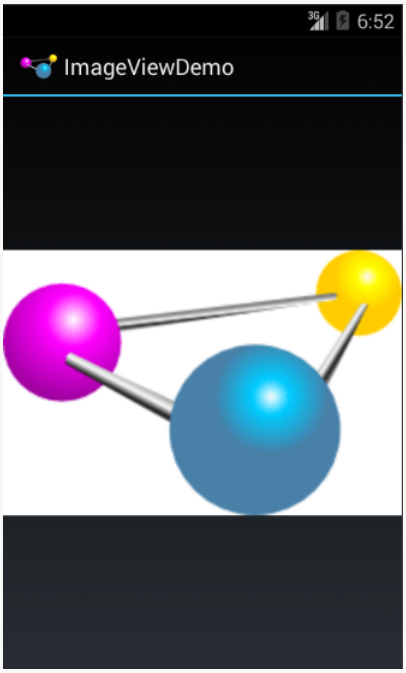


Figure 92: The ImageViewDemo sample application

Android Studio Graphical Layout Editor

The `ImageView` widget can be found in the "Images & Media" portion of the Palette in the Android Studio graphical layout editor:

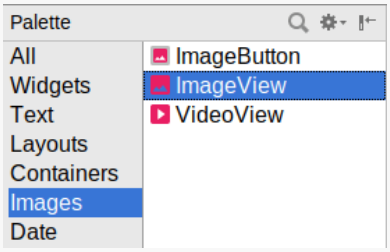


Figure 93: Widgets Palette, ImageView Shown Highlighted

`ImageButton` appears alongside `ImageView` in that tool palette.

When you drag one of these into the preview or blueprint, you are immediately greeted by a dialog to choose a drawable resource or color to use for the image:

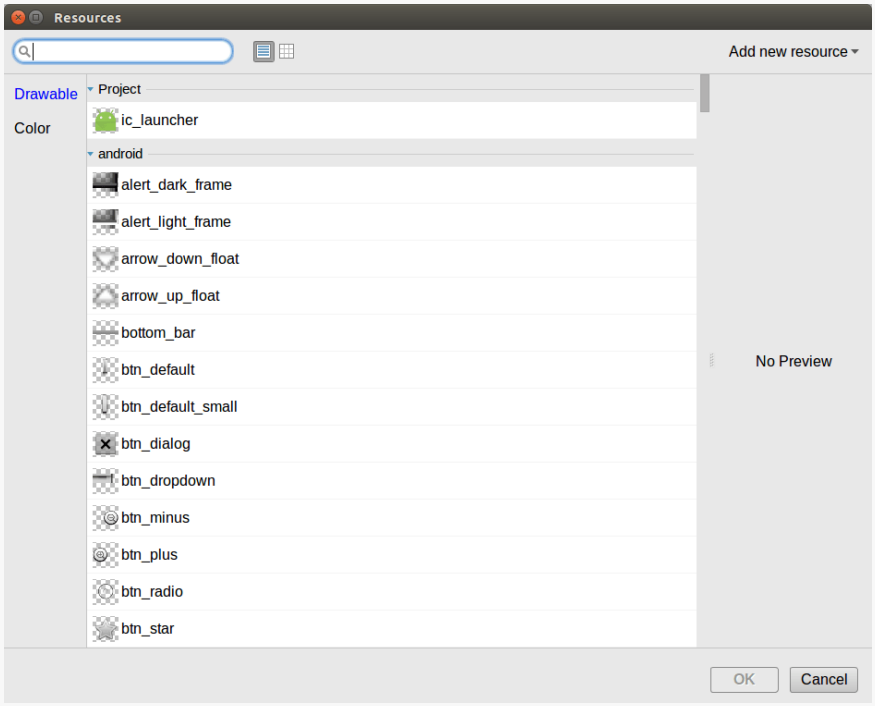


Figure 94: Image Resource Dialog

Unfortunately, you have no choice but to choose one of these, as due to a bug, if you click Cancel to exit the dialog, it also abandons the entire drag-and-drop operation.

You can drag these into a layout file, then use the Attributes pane to set their attributes. Like all widgets, you will have an "id" option to set the `android:id` value for the widget. Two others of importance, though, are more unique to `ImageView` and `ImageButton`:

- "src" allows you to choose a drawable resource to use as the image to be displayed, which will be filled in by whatever you chose in the resource dialog
- "contentDescription" provides the text that will be used to describe the image to users that have accessibility services enabled (e.g., TalkBack), such as visually impaired users
- "scaleType" opens a drop-down menu where you can choose how the image is to be scaled:

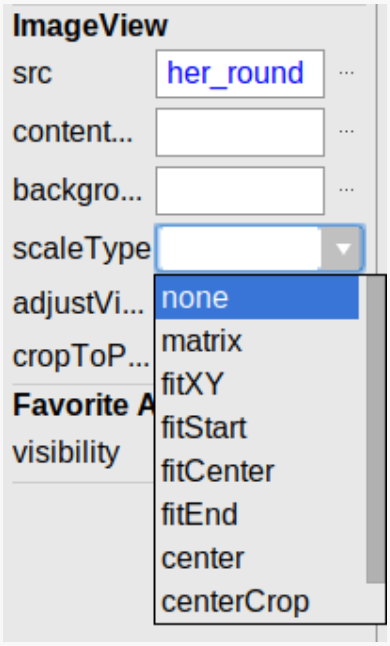


Figure 95: Scale Types in Android Studio Attributes Pane

We will examine those scale types more in the next section.

Scaling Images

It is possible, perhaps even probable, that our `ImageView` size will not exactly match the size of the images that we are trying to display. `ImageView` supports a variety of "scale types" that indicate how Android should try to deal with the discrepancy between the size/aspect ratio of the image and the size/aspect ratio of the `ImageView` itself.

These values can be seen in the JavaDocs in the `ImageView.ScaleType` class. The default (`fitCenter`) simply scales up the image to best fit the available space.

Of note, a choice of "center" will center the image in the available space but will not scale up the image:

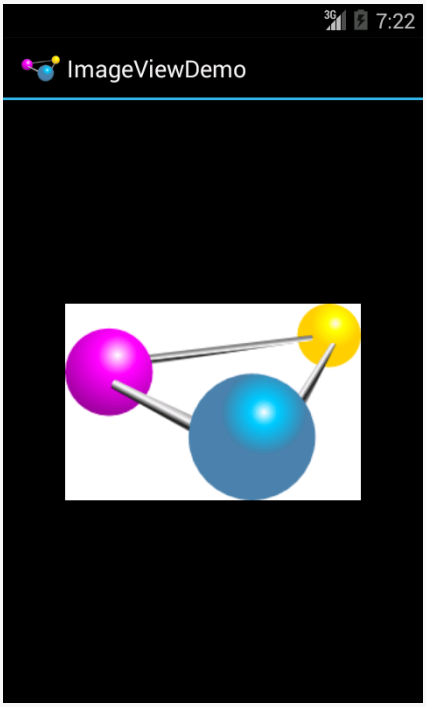


Figure 96: The ImageViewDemo Sample, Set to center

A choice of `centerCrop` will scale the image so that its shortest dimension fills the available space and crops the rest:

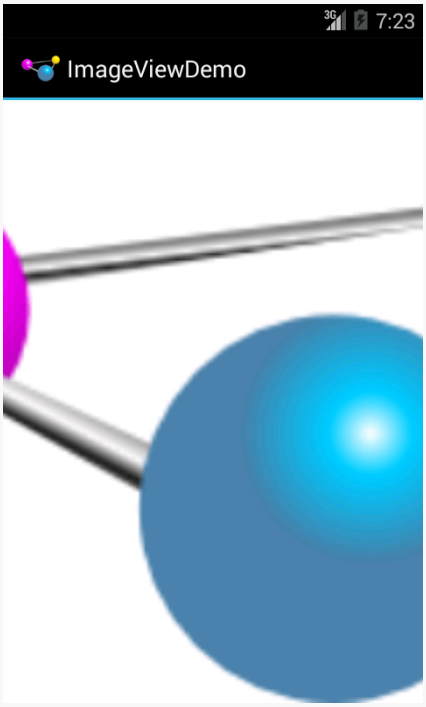


Figure 97: The ImageViewDemo Sample, Set to centerCrop

A choice of `fitXY` will scale the image to fill the space, ignoring the aspect ratio:

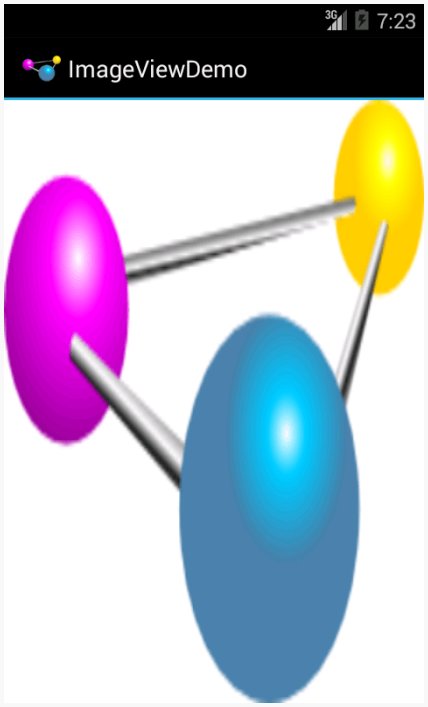


Figure 98: The ImageViewDemo Sample, Set to fitXY

Fields of Green. Or Other Colors.

Along with buttons and labels, fields are the third "anchor" of most GUI toolkits. In Android, they are implemented via the `EditText` widget, which is a subclass of the `TextView` used for labels.

Along with the standard `TextView` attributes (e.g., `android:textStyle`), `EditText` has others that will be useful for you in constructing fields, notably `android:inputType`, to describe what sort of input your `EditText` expects (numbers? email addresses? phone numbers?). A thorough explanation of `android:inputType` and its interaction with input method editors (a.k.a., "soft keyboards") will be discussed in an upcoming chapter.

For example, from the `Basic/Field` sample project, here is an XML layout file showing an `EditText`:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/field"
  android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
android:inputType="textMultiLine"
android:text="@string/license"
/>
```

(from Basic/Field/app/src/main/res/layout/main.xml)

Note that we have `android:inputType="textMultiLine"`, so users will be able to enter in several lines of text. We also have defined the initial text to be the value of a `license` string resource.

The result, once built and installed into the emulator, is:

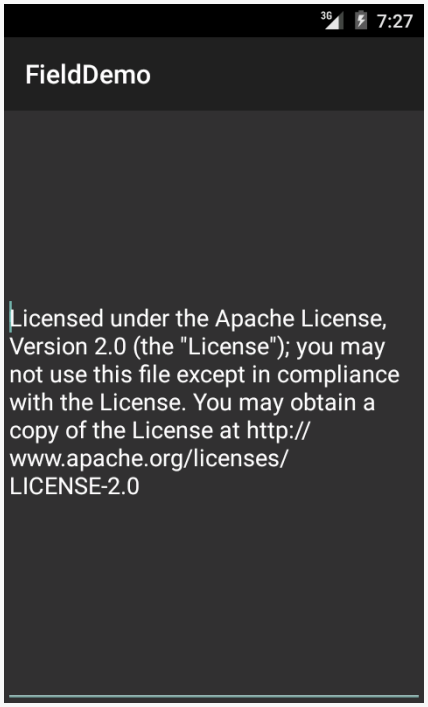


Figure 99: FieldDemo, in Theme.Material

Android Studio Graphical Layout Editor

The Android Studio Graphical Layout's Palette has a whole section dedicated primarily to `EditText` widgets, named "Text Fields (EditText)":

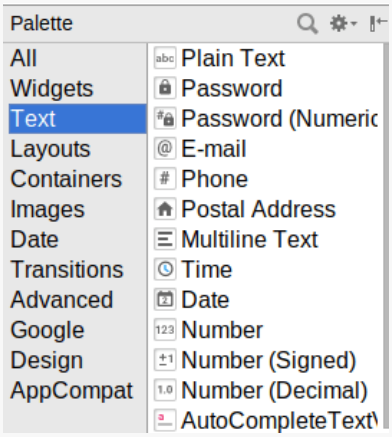


Figure 100: Text Fields Palette

The first entry is a general-purpose `EditText`. The rest come pre-configured for various scenarios, such as a password.

You can drag any of these into your layout, then use the Attributes pane to configure relevant attributes. The "Id" and "Text" attributes are the same as found on `TextView`, as are many other attributes, inherited from `TextView`.

Notable EditText Attributes

The "Hint" item in the Attributes pane allows you to set a "hint" for this `EditText`. The "hint" text will be shown in light gray in the `EditText` widget when the user has not entered anything yet. Once the user starts typing into the `EditText`, the "hint" vanishes. This might allow you to save on screen space, replacing a separate label `TextView`.

The "Input Type" item in the Attributes pane allows you to describe what sort of input you are expecting to receive in this `EditText`, lining up with many of the types of fields you can drag from the Palette into the layout:

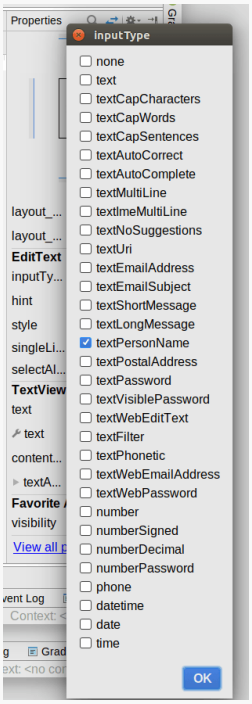


Figure 101: Android Studio's Text Fields InputType List

The `inputType` attribute will be covered in greater detail in an upcoming chapter.

More Common Concepts

All widgets, including the ones shown above, extend `View`. The `View` base class gives all widgets an array of useful attributes and methods beyond those already described.

Padding

Widgets have a minimum size, one that may be influenced by what is inside of them. So, for example, a `Button` will expand to accommodate the size of its caption. You can control this size using padding. Adding padding will increase the space between the contents (e.g., the caption of a `Button`) and the edges of the widget.

Padding can be set once in XML for all four sides (`android:padding`) or on a per-side basis (`android:paddingLeft`, etc.). Padding can also be set in Java via the `setPadding()` method. On Android 8.0+, there is also `android:paddingHorizontal` and `android:paddingVertical`, to set the padding on both sides of a single axis.

The value of any of these is a dimension — a combination of a unit of measure and a count. So, `10dip` is 10 density-independent pixels, `2mm` is 2 millimeters, etc.

Margins

By default, widgets are tightly packed, one next to the other. You can control this via the use of margins, a concept that is reminiscent of the padding described previously.

The difference between padding and margins comes in terms of the background. For widgets with a transparent background — like the default look of a `TextView` — padding and margins have similar visual effect, increasing the space between the widget and adjacent widgets. However, for widgets with a non-transparent background — like a `Button` — padding is considered inside the background while margins are outside. In other words, adding padding will increase the space between the contents (e.g., the caption of a `Button`) and the edges, while adding margin increases the empty space between the edges and adjacent widgets.

Margins can be set in XML, either on a per-side basis (e.g., `android:layout_marginTop`) or on all sides via `android:layout_margin`. Once again, the value of any of these is a dimension — a combination of a unit of measure and a count, such as `5dp` for 5 density-independent pixels. On Android 8.0+, there is also `android:layout_marginHorizontal` and `android:layout_marginVertical`, to set the margin on both sides of a single axis.

Colors

There are two types of color attributes in Android widgets. Some, like `android:background`, take a single color (or a drawable to serve as the background). Others, like `android:textColor` on `TextView` (and subclasses) can take a `ColorStateList`, including via the Java setter (in this case, `setTextColor()`).

A `ColorStateList` allows you to specify different colors for different conditions. For example, when you get to selection widgets in an upcoming chapter, you will see how a `TextView` has a different text color when it is the selected item in a list compared to when it is in the list but not selected. This is handled via the default `ColorStateList` associated with `TextView`.

If you wish to change the color of a `TextView` widget in Java code, you have two main choices:

- Use `ColorStateList.valueOf()`, which returns a `ColorStateList` in which all states are considered to have the same color, which you supply as the parameter to the `valueOf()` method. This is the Java equivalent of the `android:textColor` approach, to make the `TextView` always be a specific color regardless of circumstances.
- Create a `ColorStateList` with different values for different states, either via the constructor or via an XML drawable resource. This will be covered much later in the book.

Other Useful Attributes

Some additional attributes on `View` most likely to be used include:

1. `android:visibility`, which controls whether the widget is initially visible
2. `android:nextFocusDown`, `android:nextFocusLeft`, `android:nextFocusRight`, and `android:nextFocusUp`, which control the focus order if the user uses the D-pad, trackball, or similar pointing device
3. `android:contentDescription`, which is roughly equivalent to the `alt` attribute on an HTML `<img>` tag, and is used by accessibility tools to help people who cannot see the screen navigate the application — this is very important for widgets like `ImageView`

We will see more about the focus attributes and `android:contentDescription` in the chapter on focus management and accessibility, later in this book.

Useful Methods

You can toggle whether or not a widget is enabled via `setEnabled()` and see if it is enabled via `isEnabled()`. One common use pattern for this is to disable some widgets based on a `CheckBox` or `RadioButton` checked state. We will explore `CheckBox`, `RadioButton`, and similar sorts of widgets a bit later in the book.

You can give a widget focus via `requestFocus()` and see if it is focused via `isFocused()`. You might use this in concert with disabling widgets as mentioned above, to ensure the proper widget has the focus once your disabling operation is complete.

To help navigate the tree of widgets and containers that make up an activity's overall view, you can use:

1. `getParent()` to find the parent widget or container

2. `findViewById()` to find a child widget with a certain ID
3. `getRootView()` to get the root of the tree (e.g., what you provided to the activity via `setContentView()`)

Visit the Trails!

You can learn more about Android's input method framework — what you might think of as soft keyboards — in a later chapter.

Another chapter in the trails covers the use of fonts, to tailor your `TextView` widgets (and those that inherit from them, like `Button`).

Yet another chapter in the trails covers rich text formatting, both for presenting formatted text in a `TextView` (e.g., inline **boldface**) and for collecting formatted text from the user via a customized `EditText`.

Debugging Your App

Now that we are starting to manipulate layouts and Java code more significantly, the odds increase that we are going to somehow do it wrong, and our app will crash.

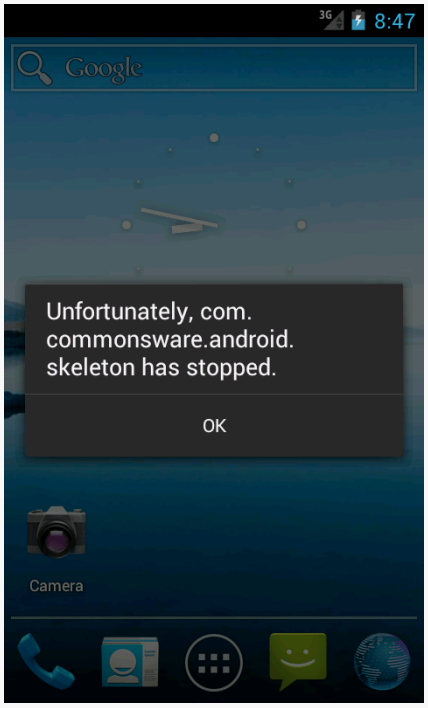


Figure 102: A Crash Dialog on Android 4.0.3

In this chapter, we will cover a few tips on how to debug these sorts of issues.

Get Thee To a Stack Trace

If you see one of those "Force Close" or "Has Stopped" dialogs, the first thing you will want to do is examine the Java stack trace that is associated with this crash. These are logged to a facility known as Logcat, on your device or emulator.

To view Logcat, you have three choices:

1. Use the `adb logcat` command at the command line (or something that uses `adb logcat`, such as various colorizing scripts available online)
2. Use the Logcat tab in the standalone Android Device Monitor utility (run `monitor` from the command line)
3. Use the Logcat view

There are also Logcat apps on the Play Store, such as aLogcat, that will display the contents of Logcat. However, for security and privacy reasons, on Android 4.1 and higher devices, such apps will only be able to show you *their* Logcat entries, not those from the system, your app, or anyone else. Hence, for development purposes, it is better to use one of the other alternatives outlined above.

Logcat in Android Studio

The Logcat view is available at any time, from pretty much anywhere in Android Studio, by means of clicking on the Android tool window entry, usually docked at the bottom of your IDE window:

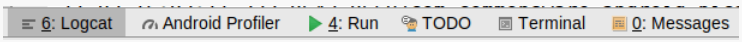


Figure 103: Minimized Tool Windows in Android Studio, Showing Logcat Tool

Tapping on that will bring up some Android-specific logs in an "Android DDMS" tool window, with a tab for "Devices | logcat":

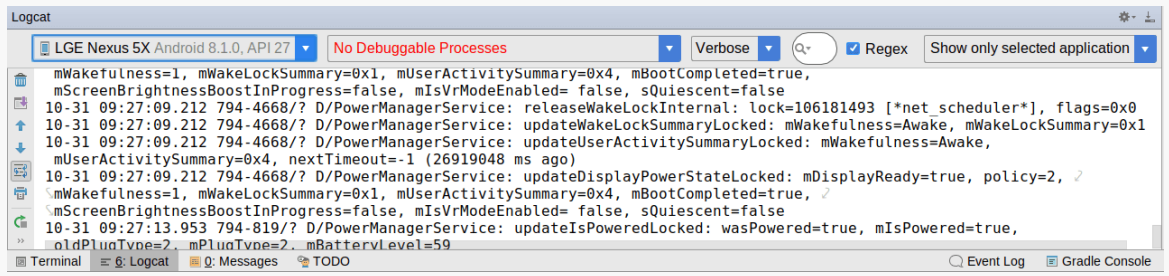


Figure 104: Android DDMS Tool Window, Showing Logcat

Logcat will show your stack traces, diagnostic information from the operating system, and anything you wish to include via calls to static methods on the `android.util.Log` class. For example, `Log.e()` will log a message at error severity, causing it to be displayed in red.

If you want to send something from Logcat to somebody else, such as via an issue tracker, just highlight the text and copy it to the clipboard, as you would with any text editor.

The "trash can" icon atop the tool strip on the left is the "clear log" tool. Clicking it will appear to clear Logcat. It definitely clears your Logcat *view*, so you will only see messages logged after you cleared it. Note, though, that this does not actually clear the logs from the device or emulator.

In addition, you can:

- Use the "Log level" drop-down to filter lines based on severity, where messages for your chosen severity or higher will be displayed
- Use the search field to the right of the "Log level" drop-down to filter items based on a search string
- Set up more permanent filters via the drop-down to the right of the search field

The Case of the Confounding Class Cast

If you crash, the stack trace might suggest that there is a problem tied to your resources. One common flavor of this is a `ClassCastException` when you call `findViewById()`. For example, you might call `(Button)findViewById(R.id.button)`, yet get a `ClassCastException: android.widget.LinearLayout` as a result,

indicating that while you thought your `findViewById()` call would return a `Button`, it really returned a `LinearLayout`.

Often times, this is not your fault. Sometimes, the `R` values get out of sync with pre-compiled classes from previous builds. This most often occurs just after you change your mix of resources (e.g., add a new layout).

To resolve this, you need to clean your project. To do this, in Android Studio, choose "Build > Clean Project" from the main menu.

So, if you get a strange crash that seems like it might be related to resources, clean your project. If the problem goes away, you are set — if the problem persists, you will need to do a bit more debugging.

Point Break

One of the hallmarks of Java IDEs is the ability to do real-time debugging, using breakpoints and the like. In that respect, Android Studio works for Android apps in the same way that IntelliJ IDEA and Eclipse work for Java apps. You can debug on an emulator or any Android device for which you enabled USB debugging (as you may have done in Tutorial #1).

Lacking any Android Studio-specific documentation, you will wind up referring to the documentation for IntelliJ IDEA to learn how to use its debugger.

With Android Studio, the run controls in the toolbar will give you some options for debugging your app:

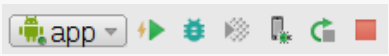


Figure 105: Android Studio Run Controls

The bug-shaped button to the right of the "run" green triangle will launch your app and attach the debugger, so breakpoints will be honored. If your app is already running, and you want to debug the running process, you can do that via the toolbar button that looks like a phone with a small bug in the lower-right corner.

The Classic Container Classes

Containers — sometimes referred to as layout managers — organize widgets on the screen. Containers position and size widgets based upon rules that you supply along with key device characteristics, such as available screen size.

Three containers have dominated Android app development since Android's introduction in late 2007: `LinearLayout`, `RelativeLayout`, and `TableLayout`. This chapter focuses on those. Later chapters will explore other container classes, such as 2016's `ConstraintLayout`.

Introducing the Sampler App

The `Containers/Sampler` sample project has a bunch of layout resources that this chapter will use to illustrate how these containers work.

If you were to run this sample app, you would see a series of tabs, with one layout displayed per tab:

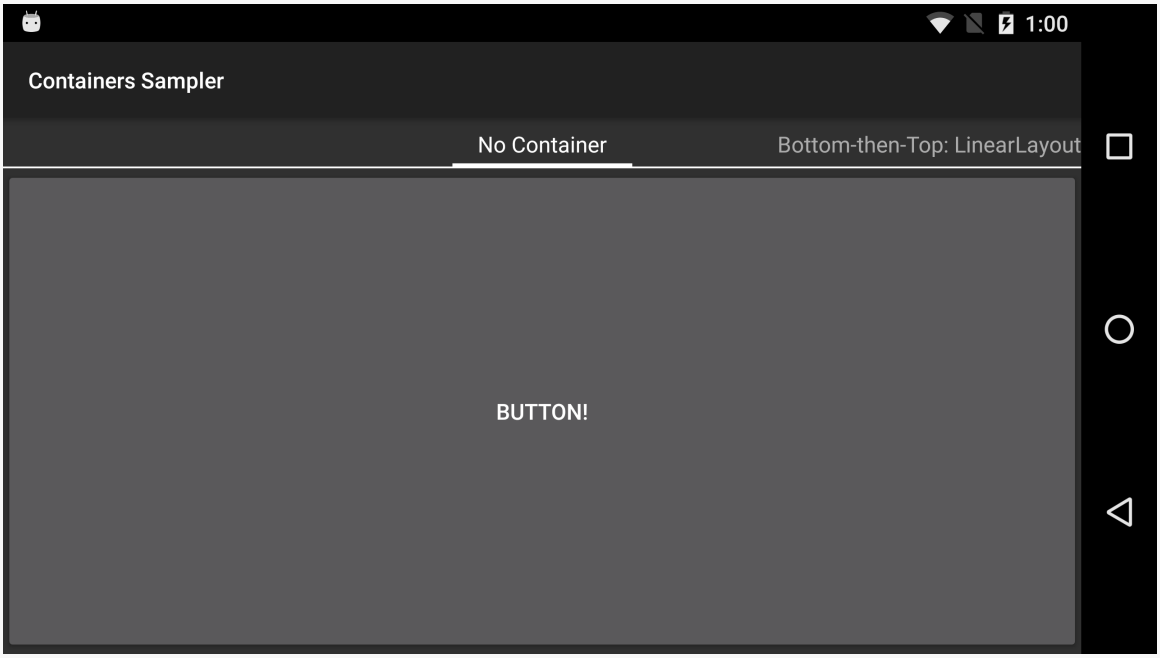


Figure 106: Sampler App, As Initially Launched

We are not going to get into the Java code associated with this sample app in this chapter. That code relies on other topics, like fragments and the `ViewPager` widget, that we have not gotten to yet. We will come back to this sample app and see how the tabs were implemented then. For now, the focus is on the layout files showing specific techniques for using these classic containers.

RTL and Your Layouts

Most of the world's languages are written left-to-right. So, in this paragraph, you read the letters and words starting from the left edge of a line across to the right edge.

Arabic and Hebrew, among others, are written right-to-left. The abbreviation "RTL" refers to these languages.

Originally, Android was focused purely on left-to-right (LTR) languages. As a result, you see attributes referring to things with respect to left and right (e.g., `android:paddingLeft`).

Slowly, Android improved its RTL support. In particular, starting with API Level 17 (Android 4.2), analogue attributes were added, replacing "left" with "start" and

"right" with "end". When using "start"/"end" attributes (e.g., `android:paddingStart`), "start" refers to where you start reading a line of text, and "end" refers to where you end reading a line of text:

Language Direction	"Start" Means...	"End" Means...
LTR	Left	Right
RTL	Right	Left

In general, we want the GUI to flow with the language direction. Things that you might have on the left with an LTR language usually go on the right with an RTL language, and so forth.

However, since these "start" and "end" attributes are new to API Level 17, you cannot use them on older devices. Sometimes, we will wind up either using the old "left"/"right" attributes or using both types of attributes, to cover all device versions.

We will revisit RTL language support a bit later in this chapter.

LinearLayout and the Box Model

`LinearLayout` represents Android's approach to a box model — widgets or child containers are lined up in a column or row, one after the next.

Concepts and Attributes

To configure a `LinearLayout`, you have four main areas of control besides the container's contents: the orientation, the fill model, the weight, the gravity.

Orientation

Orientation indicates whether the `LinearLayout` represents a row or a column. Just add the `android:orientation` property to your `LinearLayout` element in your XML layout, setting the value to be `horizontal` for a row or `vertical` for a column.

The orientation can be modified at runtime by invoking `setOrientation()` on the `LinearLayout`, supplying it either `HORIZONTAL` or `VERTICAL`.

Fill Model

The point behind a `LinearLayout` — or any of the Android container classes – is to organize multiple widgets. Part of organizing those widgets is determining how much space each gets.

`LinearLayout` takes an "eldest child wins" approach towards allocating space. So, if we have a `LinearLayout` with three children, the first child will get its requested space. The second child will get its requested space, if there is enough room remaining, and likewise for the third child. So if the first child asks for all the space (e.g., this is a `horizontal` `LinearLayout` and the first child has `android:layout_width="match_parent"`), the second and third children will wind up with zero width.

Weight

But, what happens if we have two or more widgets that should split the available free space? For example, suppose we have two multi-line fields in a column, and we want them to take up the remaining space in the column after all other widgets have been allocated their space.

To make this work, in addition to setting `android:layout_width` (for rows) or `android:layout_height` (for columns), you must also set `android:layout_weight`. This property indicates what proportion of the free space should go to that widget. If you set `android:layout_weight` to be the same non-zero value for a pair of widgets (e.g., `1`), the free space will be split evenly between them. If you set it to be `1` for one widget and `2` for another widget, the second widget will use up twice the free space that the first widget does. And so on.

The weight for a widget is zero by default.

Another pattern for using weights is if you want to allocate sizes on a percentage basis. To use this technique for, say, a horizontal layout:

1. Set all the `android:layout_width` values to be `0` for the widgets in the layout
2. Set the `android:layout_weight` values to be the desired percentage size for each widget in the layout
3. Make sure all those weights add up to `100`

If you want to have space left over, not allocated to any widget, you can add an `android:weightSum` attribute to the `LinearLayout`, and ensure that the sum of the

android:layout_weight attributes of the children are less than that sum. The children will each get space allocated based upon the ratio of their android:layout_weight compared to the android:weightSum, not compared to the sum of the weights. And there will be empty space that takes up the rest of the room not allocated to the children.

Gravity

By default, everything in a LinearLayout is start- and top-aligned. So, if you create a row of widgets via a horizontal LinearLayout, the row will be flush on the start side of the screen (e.g., left in a LTR language).

If that is not what you want, you need to specify a gravity. Unlike the physical world, Android has two types of gravity: the gravity of a widget within a LinearLayout, and the gravity of the contents of a widget or container.

The android:gravity property of some widgets and containers — which also can be defined via setGravity() in Java — tells Android to slide the contents of the widget or container in a particular direction. For example, android:gravity="right" says to slide the contents of the widget to the right; android:gravity="right|bottom" says to slide the contents of the widget to the right and the bottom.

Here, "contents" varies. TextView supports android:gravity, and the "contents" is the text held within the TextView. LinearLayout supports android:gravity, and the "contents" are the widgets inside the container. And so on.

Children of a LinearLayout also have the option of specifying android:layout_gravity. Here, the child is telling the LinearLayout "if there is room, please slide me (and me alone) in this direction". However, this only works in the direction *opposite* the orientation of the LinearLayout – the children of a vertical LinearLayout can use android:layout_gravity to control their positioning horizontally (start or end), but not vertically.

For a row of widgets, the default is for them to be aligned so their texts are aligned on the baseline (the invisible line that letters seem to "sit on"), though you may wish to specify a gravity of center_vertical to center the widgets along the row's vertical midpoint.

Android Studio Graphical Layout Editor

The `LinearLayout` container can be found in the "Layouts" portion of the Palette of the Android Studio graphical layout editor:

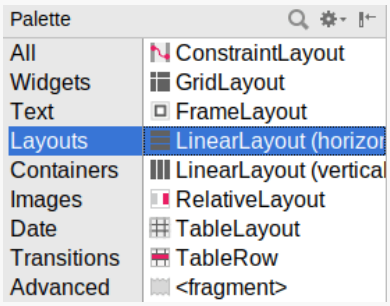


Figure 107: Layouts Palette in Android Studio Graphical Layout Editor

You can drag either the "LinearLayout (vertical)" or "LinearLayout (horizontal)" into a layout XML resource, then start dragging in children to go into the container.

When your `LinearLayout` is the selected widget, a few new toolbar buttons will appear over the preview:

Figure 108: LinearLayout Toolbar Buttons

The only one of these that works is the left hand one, which toggles the `LinearLayout` between `vertical` and `horizontal` orientation.

When one of the *children* of the `LinearLayout` is the selected widget, the toolbar changes:

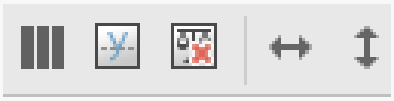


Figure 109: LinearLayout Toolbar Buttons, For Selected Child

From left to right, the buttons:

- Toggle the parent `LinearLayout` between horizontal and vertical orientations
- Align the child widgets' baselines (where a "baseline" is the invisible line that text appears to sit upon)
- Removes the `android:layout_weight` attribute, if it has one
- Toggle the width between `match_parent` and `wrap_content`
- Toggle the height between `match_parent` and `wrap_content`

Example: Bottom-then-Top

Those rules will make more sense once we work through some examples.

The first example is where we have two widgets. One widget should take up its "natural" amount of space, anchored to the bottom of the screen. The other widget should fill all the remaining space:

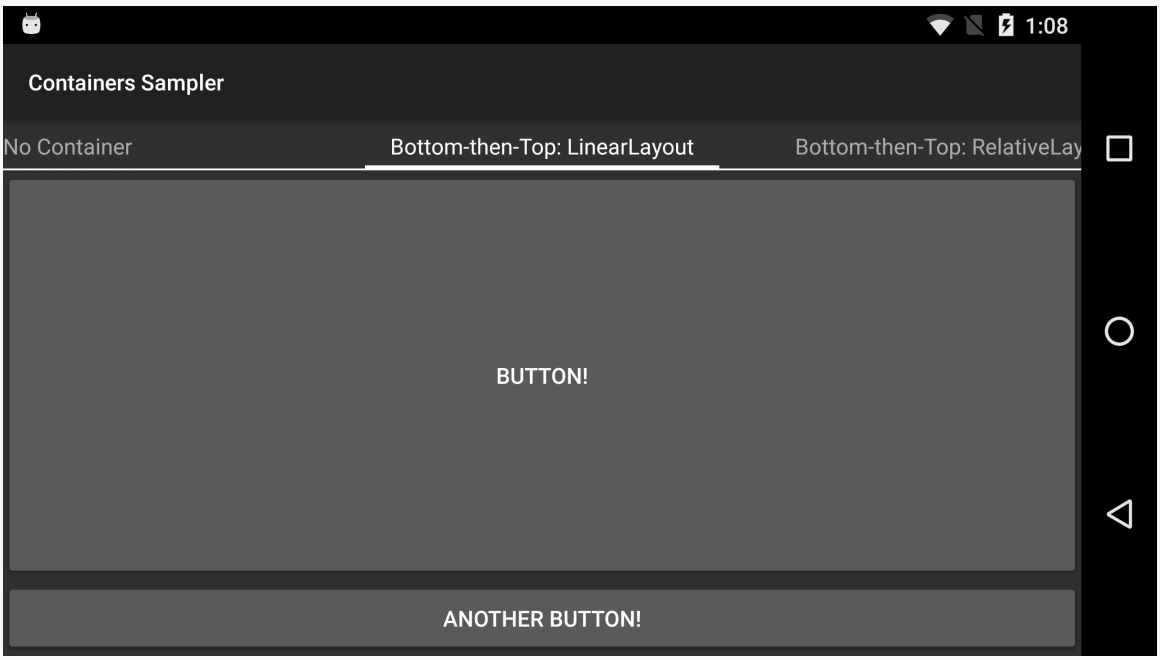


Figure 110: Bottom-then-Top Layout, Using LinearLayout

The XML

The layout XML that generated that screenshot consists of a `vertical` `LinearLayout` and two `Button` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:baselineAligned="false">

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:text="@string/button" />

  <Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/another_button" />
</LinearLayout>
```

(from Containers/Sampler/app/src/main/res/layout/bottom_then_top_ll.xml)

The `LinearLayout` has its sizes (`android:layout_width`, `android:layout_height`) set each to `match_parent`, so it will fill up all space available to it. In this sample app, that will be everything below the tabs.

Both `Button` widgets have their `android:layout_width` attributes also set to `match_parent`, so they will fill up all available space on the screen horizontally.

The bottom `Button` has its `android:layout_height` set to `wrap_content`, so it will only take up as much space as is needed to render its caption and background around it.

The top `Button` has `android:layout_height` set to `0dp`. If we did not do anything else, that would result in an impossibly-short button. However, we also have `android:layout_weight="1"`.

When this gets rendered on the screen, `LinearLayout` will make two passes through its children. On the first pass, it asks for how much space each child wants, based upon the `android:layout_height` values (since this is a `vertical` `LinearLayout`). The first `Button` will ask for 0 pixels of space, while the bottom `Button` will ask for however much it needs for its content. Then, the second pass of the `LinearLayout` through its children asks for their weights. The first `Button` has a weight of 1, while the bottom `Button` has the default weight of 0. As a result, the top `Button` gets 1/1 =

100% of all pixels left over from what the first pass used. This causes the top `Button` to become tall, shoving the bottom `Button` to the bottom of the screen.

Android Studio Graphical Layout Editor

You can create a new layout resource by right-clicking over the `res/layout/` directory and choosing "New > Layout resource file" from the right-mouse context menu. You then get a dialog where you can give the new resource a name and choose the root container for that layout:

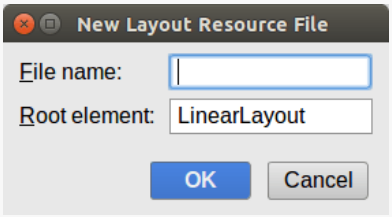


Figure 111: New Layout Resource Dialog

What you get by default depends a bit on your project:

- If your project has `ConstraintLayout` available to it, the default will be a `ConstraintLayout`
- Otherwise, the default will be a `vertical` `LinearLayout`

As you drag `Button` widgets into the layout, they are initially given a width of `match_parent` and a height of `wrap_content`:

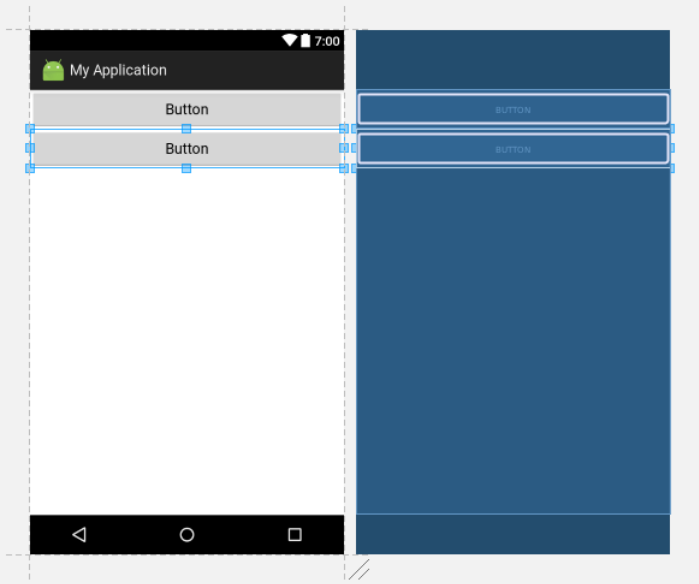


Figure 112: Vertical LinearLayout, with Two Button Widgets

Clicking the top button, then setting the `layout_weight` attribute to `1` in the Attributes pane, gives the desired look:

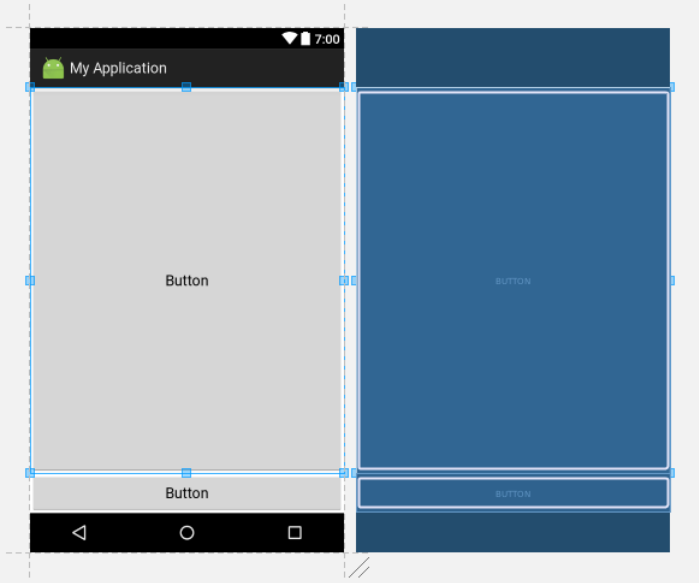


Figure 113: Vertical LinearLayout, with Weighted Button Widgets

Note that `layout_weight` may only appear for you in the "View all attributes" perspective of the Attributes pane, rather than in the simplified list.

Example: Stacked-Percent

Weights can get more elaborate than just giving all extra room to one widget.

The stacked-percent scenario allocates space on a percentage basis to all of the children of the `LinearLayout`. In the sample app, we have a layout file with three buttons, taking up 50%, 30%, and 20% of the space in a `vertical` `LinearLayout`:

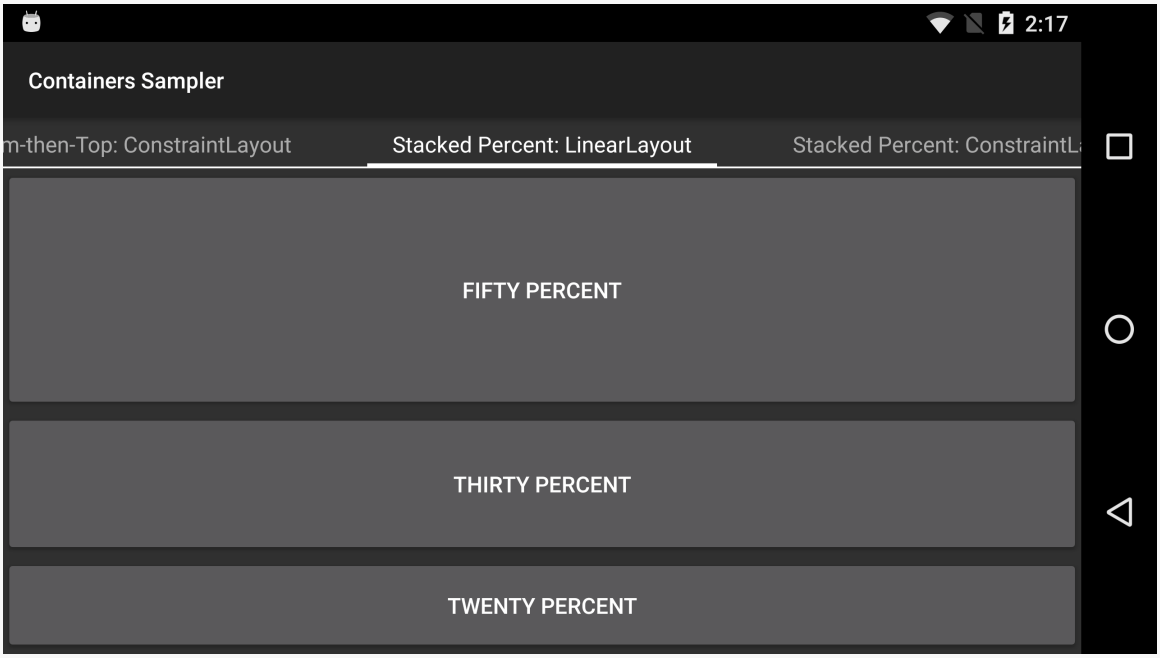


Figure 114: Stacked-Percent Layout, Using LinearLayout

The XML

This time, the layout resource XML has all three `Button` widgets with heights of 0:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="50"
```

```
    android:text="@string/fifty_percent"/>

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="30"
    android:text="@string/thirty_percent"/>

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="20"
    android:text="@string/twenty_percent"/>

</LinearLayout>
```

(from Containers/Sampler/app/src/main/res/layout/stacked_percent_ll.xml)

So, the first pass that the `LinearLayout` makes through its children, each child asks for 0 pixels of height, meaning that all of the `LinearLayout` space is available for the second pass.

On the second pass, the children report weights of 50, 30, and 20, respectively. 50+30+20=100, so the top `Button` gets 50/100 (50%) of the space, the middle `Button` gets 30/100 (30%) of the space, and the bottom `Button` gets 20/100 (20%) of the space.

Note that you would get the same results with weights of 5, 3, and 2. Really, it is the ratios of the weights that matter. However, if you are used to thinking in terms of percentages — perhaps due to past experience with other GUI toolkits — you can use integer percentages if you want.

Android Studio Graphical Layout Editor

Constructing this using the graphical layout editor is similar to the previous example:

- Create the new layout resource, choosing a `LinearLayout` as the root container.
- Drag three `Button` widgets into the `LinearLayout`.
- Using the Attributes pane, set the `android:layout_height` values on the three `Button` widgets to `0dp`

- Using the Attributes pane, set the `android:layout_weight` values to the three `Button` widgets. Note, as before, that the weight may not be on the condensed attributes list. You may need to view all attributes to have access to it:

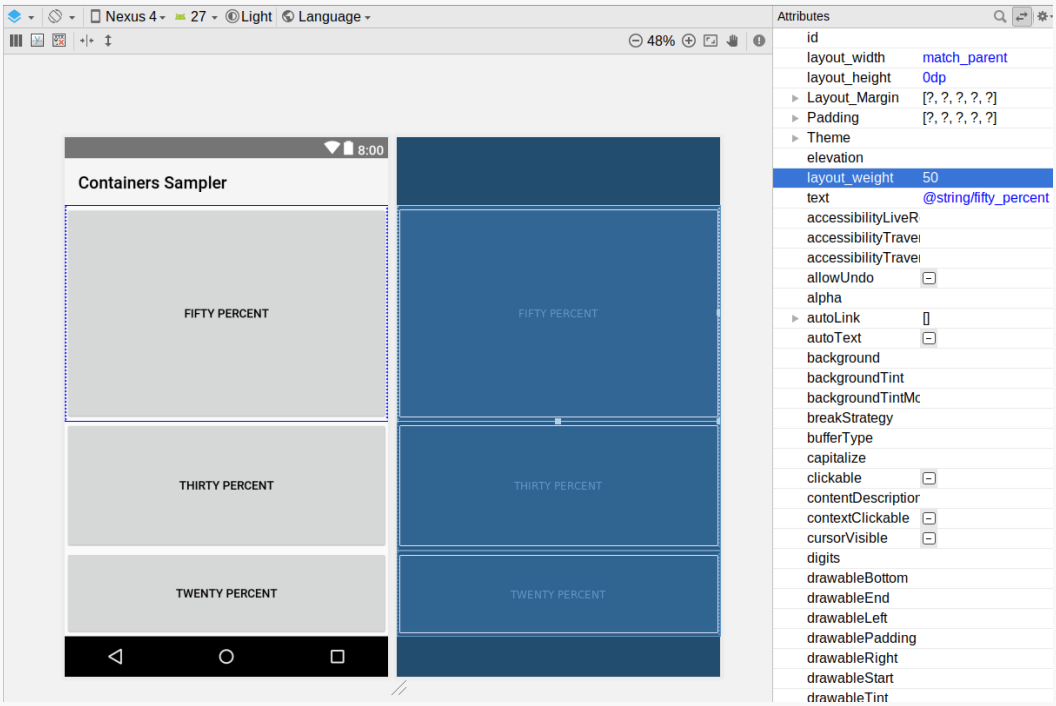


Figure 115: Vertical LinearLayout, with Full Attributes List

Example: URL Dialog

Of course, you are not limited to one axis. You can nest `LinearLayout` widgets to structure things along both the X and the Y axis.

For example, you might be aiming for a dialog-style form like this:

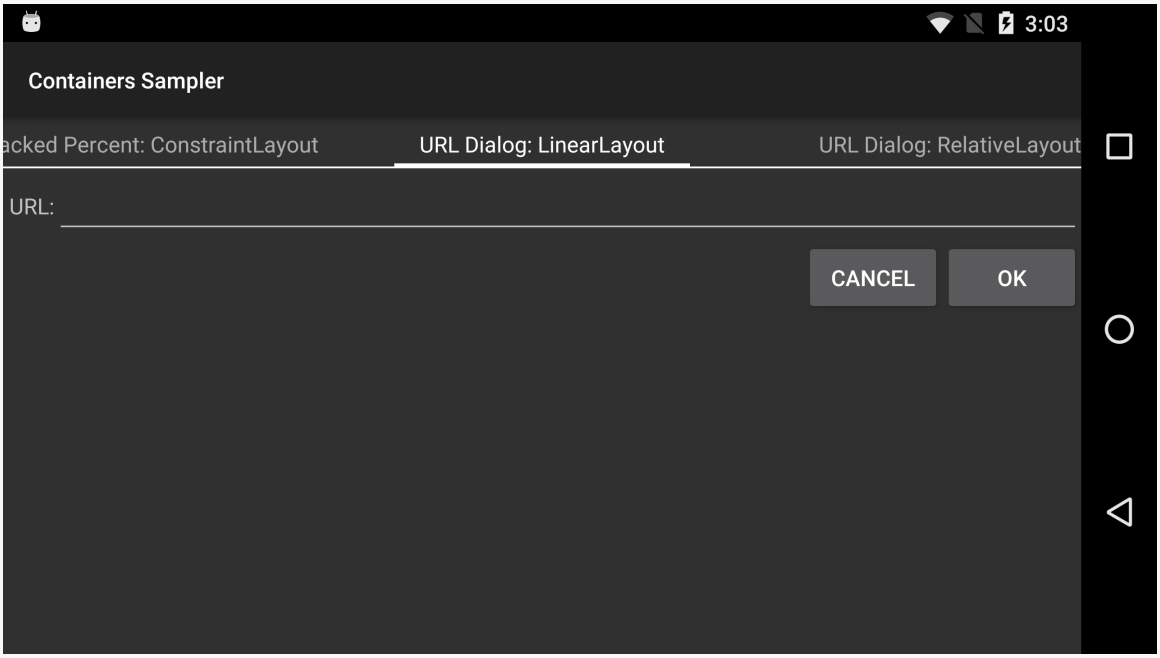


Figure 116: URL-Dialog Layout, Using LinearLayout

The XML

This layout resource is a bit more complicated, as we are now up to four widgets plus three `LinearLayout` containers:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginLeft="4dp"
      android:layout_marginStart="4dp"
      android:text="@string/url" />

    <EditText
```

```
      android:id="@+id/entry"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text" />

  </LinearLayout>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="end"
    android:orientation="horizontal">

    <Button
      android:id="@+id/cancel"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/cancel" />

    <Button
      android:id="@+id/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/ok" />
  </LinearLayout>

</LinearLayout>
```

(from Containers/Sampler/app/src/main/res/layout/url_dialog_ll.xml)

The outer `LinearLayout` is `vertical`, serving to stack the two `horizontal` `LinearLayout` rows. Each of those `horizontal` `LinearLayout` containers has a height of `wrap_content` and a width of `match_parent`, so they span the width of the screen but only need as much height as is required to render their widget contents.

The first of our four widgets is the `TextView`. Since it is in the first `horizontal` `LinearLayout`, it will be flush on the one side (left in a LTR language like English). However, we have `4dp` of margin, using both the older `android:layout_marginLeft` and the newer, RTL-capable `android:layout_marginStart` attributes. So, the widget is inset a bit from the edge.

The second widget is the `EditText`. It has a width of `match_parent`, so it will take over all remaining space after the `TextView`.

The bottom `LinearLayout` has `android:gravity="end"`, which will cause its contents to slide towards the end side of the `LinearLayout` (right in a LTR language like English). It contains the two `Button` widgets, and that is why the two `Button` widgets are slid over to the opposite end of the form.

Android Studio Graphical Layout Editor

Not surprisingly, the more complex the layout you want to create, the more work is required to create it. The IDE only helps to a point.

So, to construct this layout, you would need to:

- Create the new layout resource, choosing a `LinearLayout` as the root container.
- Drag a "LinearLayout (horizontal)" item from the Palette into the `vertical` `LinearLayout`. This unfortunately results in the new `LinearLayout` consuming all of the space of its parent, as its height and width are both `match_parent` by default:

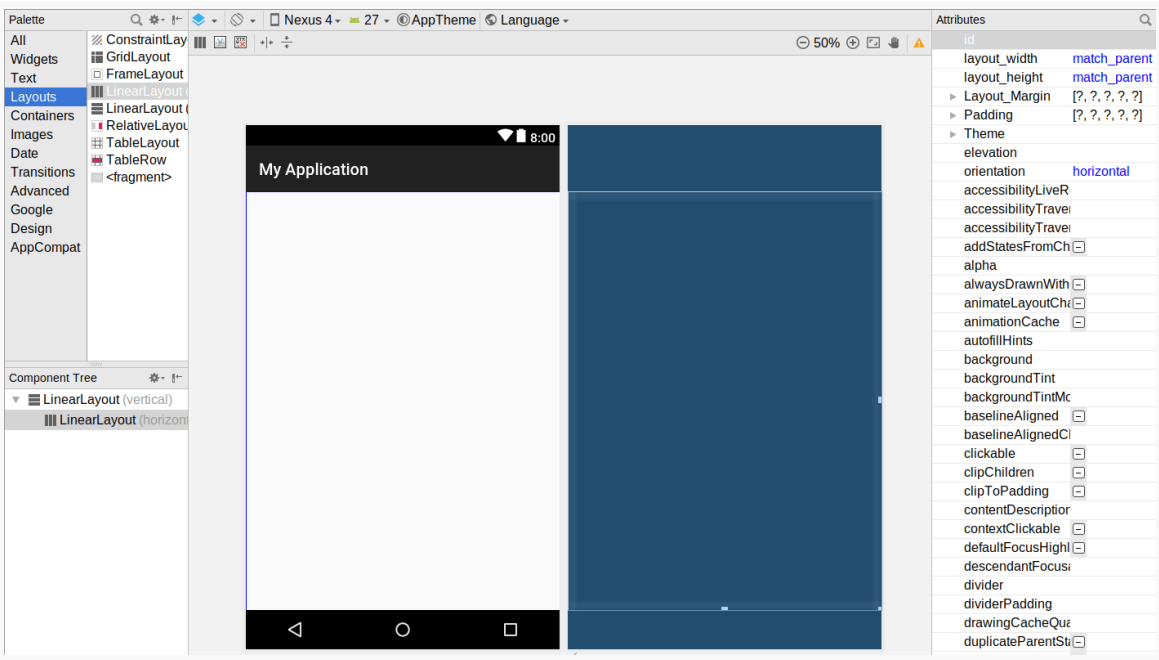


Figure 117: Vertical LinearLayout Holding Horizontal LinearLayout

- Using the Attributes pane, set the padding to be `8dp` on all sides:

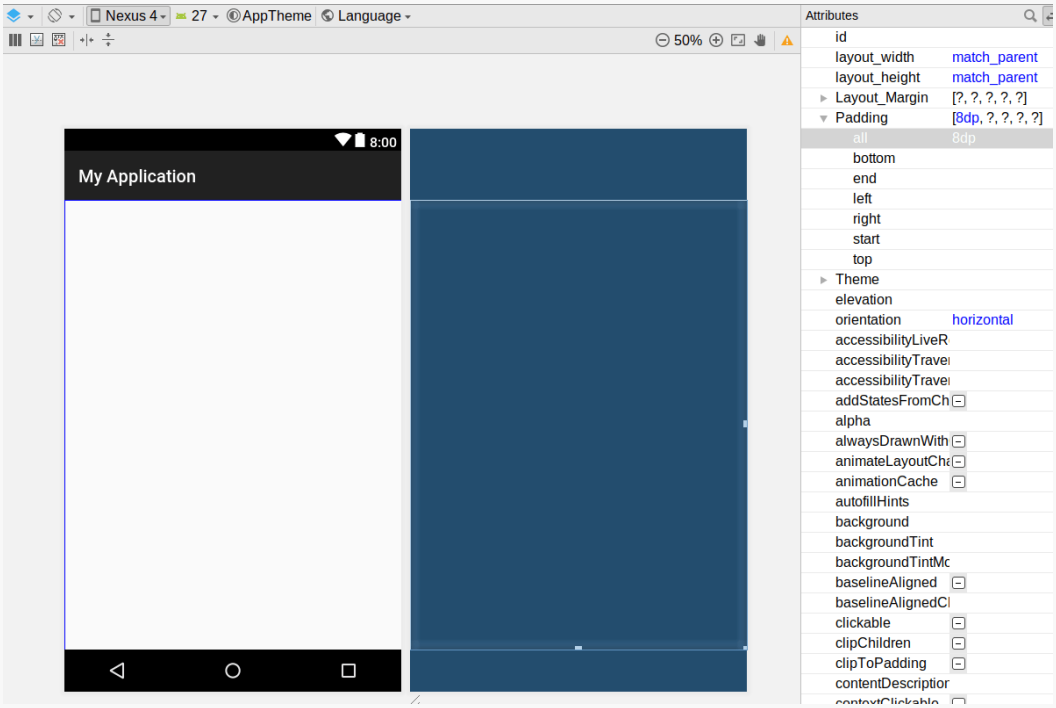


Figure 118: 8dp Padding on the Horizontal LinearLayout

- Drag a `TextView` into the `horizontal LinearLayout` *in the Component Tree*, and set its `text` to `URL:`, its width to `wrap_content`, and remove its `layout_weight` value:

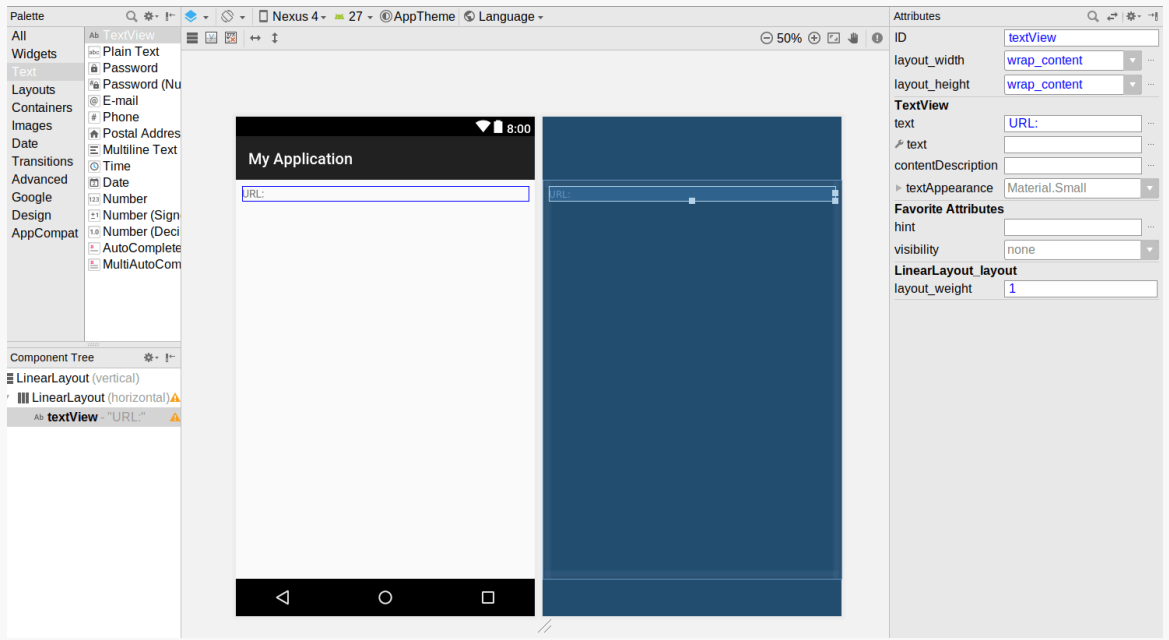


Figure 119: Horizontal LinearLayout with TextView

- Drag a "Plain Text" `EditText` into the `horizontal LinearLayout`, setting its width to be `match_parent`, its `text` to be empty, its `inputType` to be something appropriate (e.g., `textUri`, since in theory this field should hold a URL), and remove its weight. Note that you may find it easier to drag the widget into its container via the Component Tree tool, as you can better control the order of the children that way:

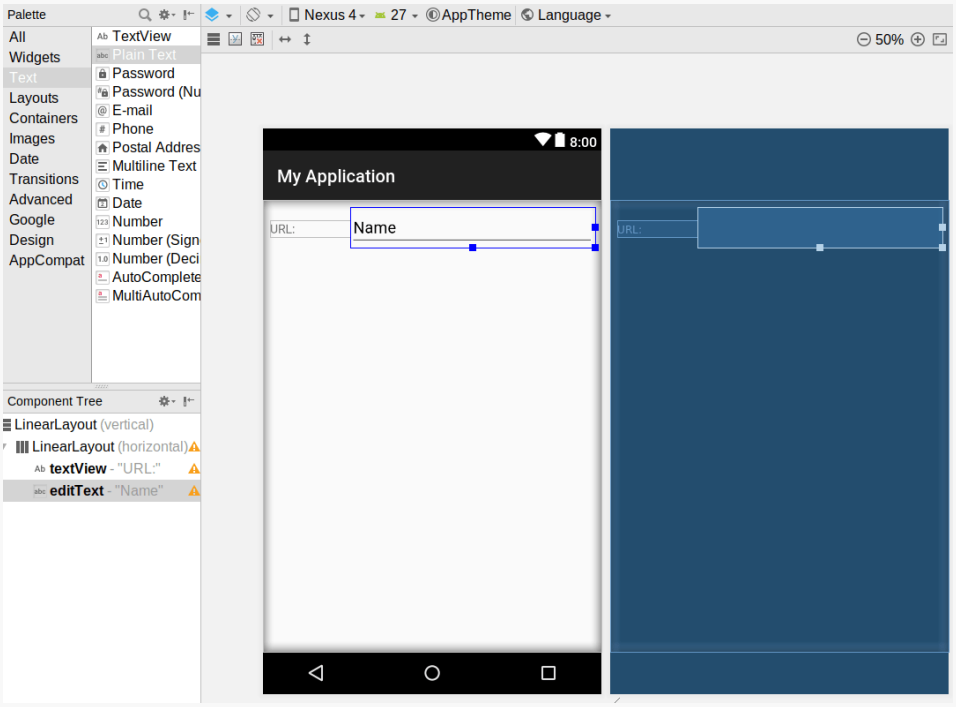


Figure 120: Horizontal LinearLayout with TextView and EditText

- Drag another "LinearLayout (horizontal)" item from the Palette, but this time drop it on the "LinearLayout(vertical)" in the Component Tree, then reordering the children in the Component Tree to put the new `LinearLayout` at the bottom:

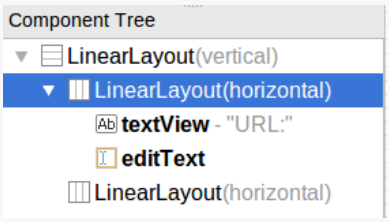


Figure 121: Component Tree with Two Horizontal LinearLayouts

- Adjust the height of both `LinearLayout` widgets to be `wrap_content`.
- Drag two `Button` widgets into that lower `LinearLayout` in the Component Tree:

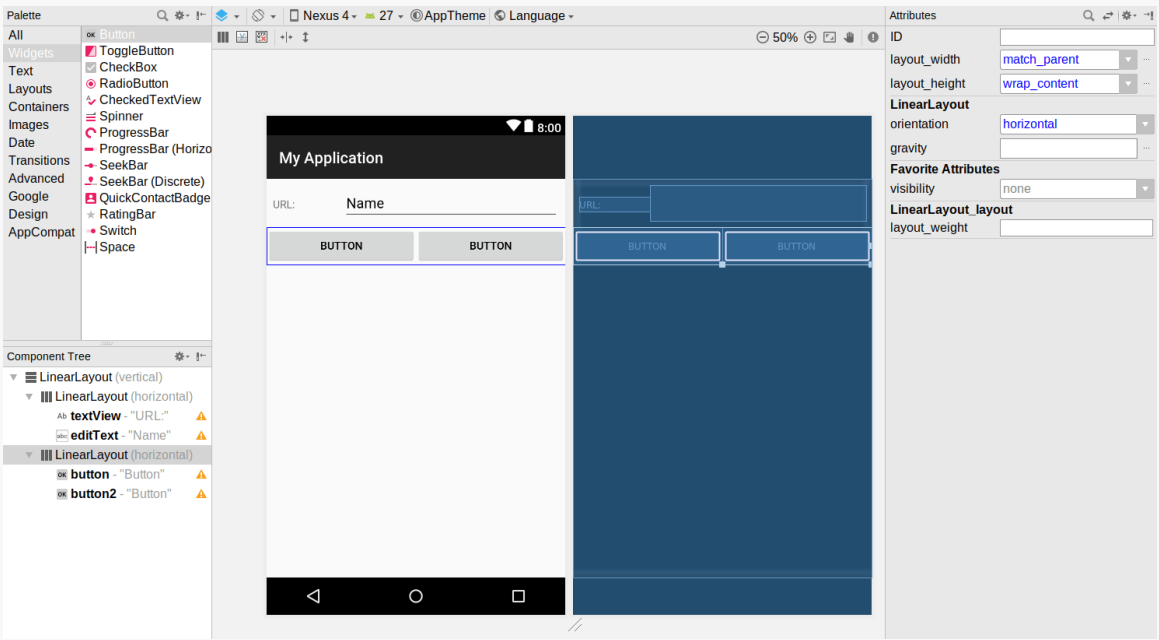


Figure 122: Two Buttons in Lower Horizontal LinearLayout

- Remove the weights from both `Button` widgets via the Attributes pane:

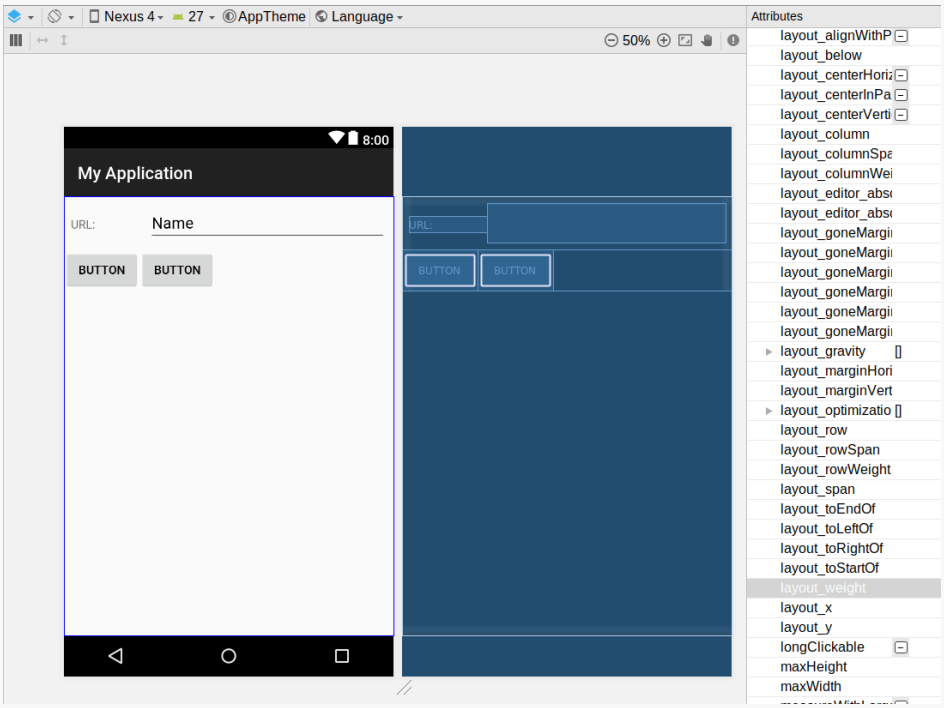


Figure 123: Two Smaller Buttons in Lower Horizontal LinearLayout

- Change the `gravity` of the lower `horizontal` `LinearLayout` to be `end`, using the Attributes pane:

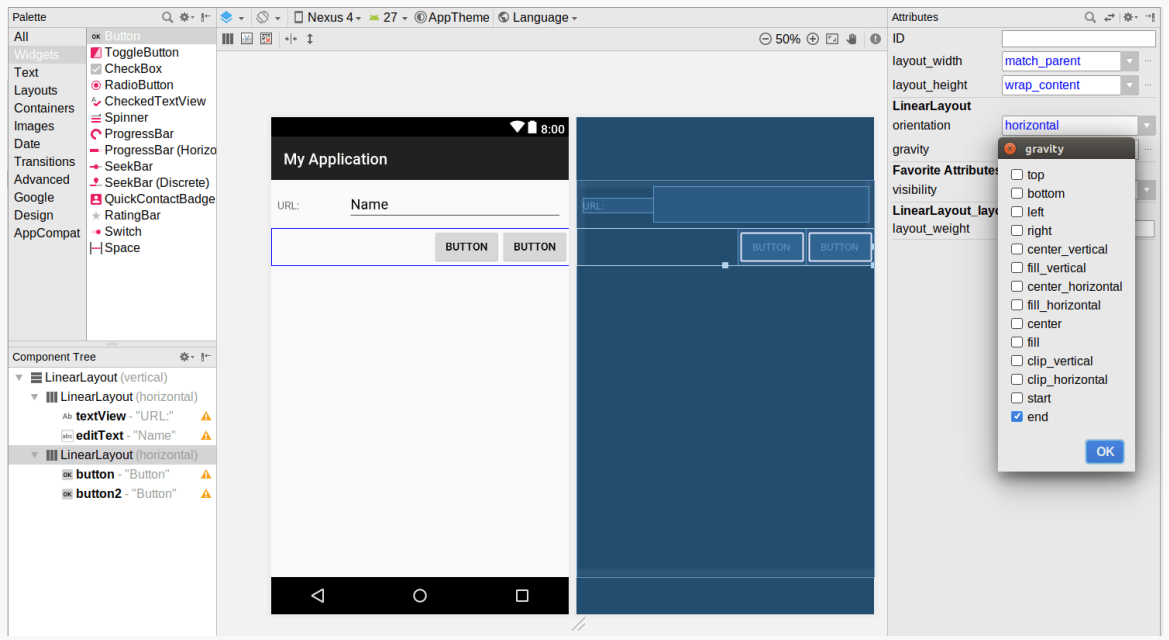


Figure 124: Two Smaller Right-Flush Buttons in Lower Horizontal LinearLayout

- Change the captions on the buttons to “OK” and “Cancel”:

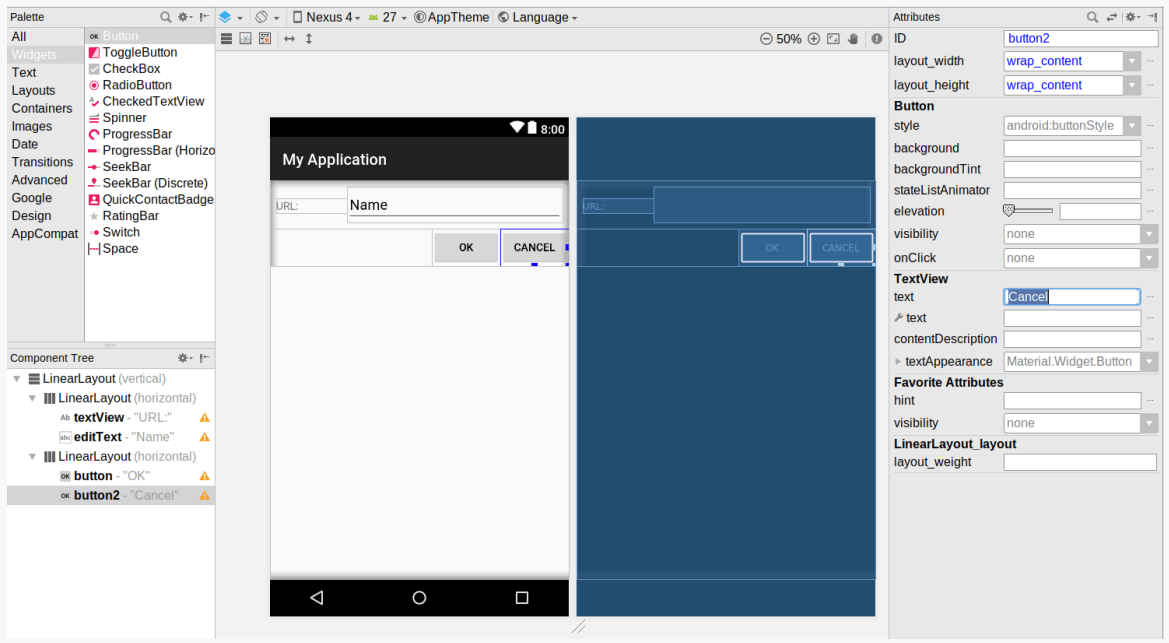


Figure 125: Buttons with Desired Captions

Now, having done all of that, the right-most toolbar button above the preview and blueprint will now be a yellow triangle, indicating some warnings about the layout that we built. Tapping that icon will display a pane with messages explaining what Android Studio does not like:

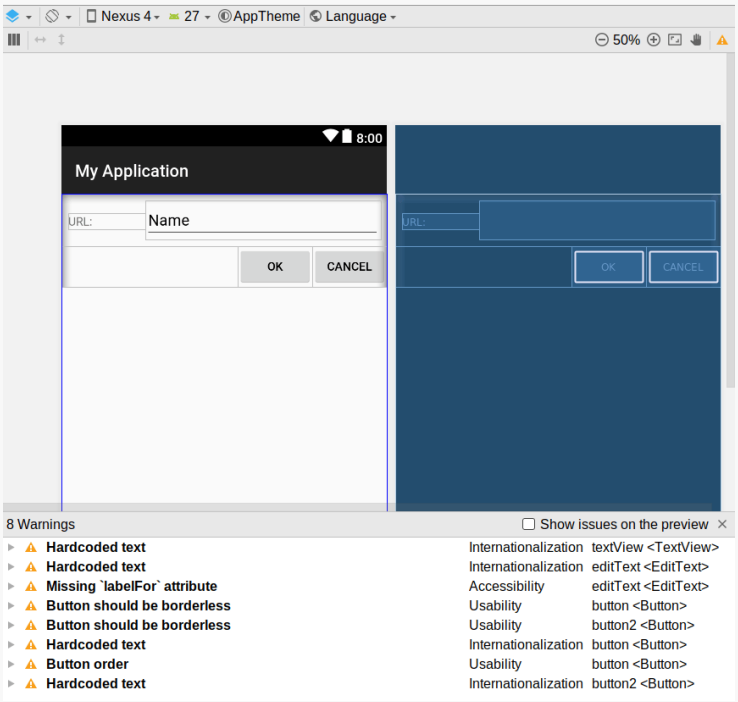


Figure 126: Layout Warning Messages

Each of the warnings in the list usually has an explanation, some suggested fixes, and other explanatory information. Clicking the button for a suggested fix will apply that fix and clear up that warning.

Some of the warnings may come from the captions for the `TextView` and `Button` widgets. If you just type a string into the `Attributes` pane, that fills in the literal string into the layout, and ideally we use string resources. For those, the "Extract string resource" suggested fix will allow you to define those as string resources.

Some of those warnings will be about certain things about styles (e.g., "Buttons in button bars should be borderless"). While in the long run you might care about those warnings, in the short term, while you are learning Android, ignoring the warnings using the suggested fix is a good idea.

One of those warnings will be about a missing `android:labelFor` attribute, which is part of the accessibility support system in Android, which we will explore later. Another is about the order of the buttons, which Google disagrees with.

Example: A Bigger Form

Not everything in a form has to be inside a `horizontal LinearLayout` which is itself inside a `vertical LinearLayout`. You only need a `horizontal LinearLayout` when you have a row that contains two or more widgets. For single-widget rows, they can just be simple children of `vertical LinearLayout`.

For example, perhaps you have a more elaborate form in mind, with several fields and other widgets, like this one:

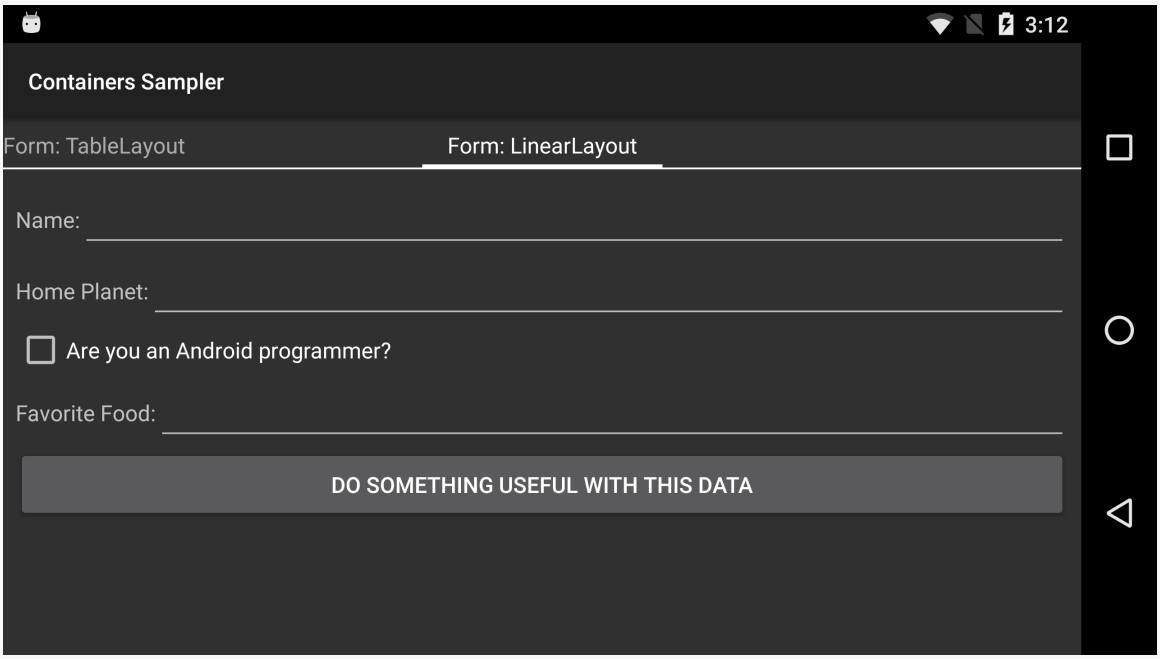


Figure 127: Form Layout, Using LinearLayout

The XML

The form's layout resource resembles the URL-dialog scenario from earlier:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:padding="8dp">

  <LinearLayout
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/name" />
    <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text" />
</LinearLayout>

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/home_planet" />
    <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text" />
</LinearLayout>

<CheckBox
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/android_programmer" />

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/favorite_food" />
    <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text" />
</LinearLayout>

<Button
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:text="@string/do_something" />

</LinearLayout>
```

(from Containers/Sampler/app/src/main/res/layout/form_ll.xml)

However, note that the `CheckBox` and the `Button` are not themselves wrapped in `horizontal` `LinearLayout` containers. They are merely direct children of the `vertical` `LinearLayout`. In this case, both are set to have a width of `match_parent`, though that is not required — you could have them set to `wrap_content` if you prefer.

We will look at `CheckBox` in greater detail in an upcoming chapter.

Android Studio Graphical Layout Editor

Setting up this form follows the same basic recipe as was used for the simpler dialog form from earlier in this chapter:

- Set up a layout resource rooted in a `vertical` `LinearLayout`
- Drag a `horizontal` `LinearLayout` into the `vertical` `LinearLayout`
- Drag a `TextView` and an `EditText` into the `horizontal` `LinearLayout` and configure as needed
- Repeat those last two steps for each of the other five rows

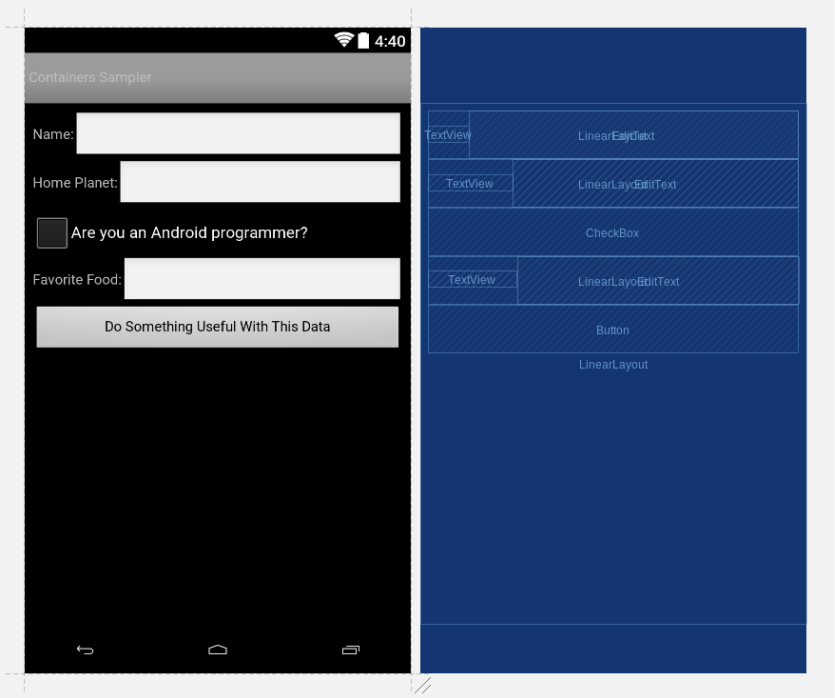


Figure 128: Bigger Form, As Seen in Android Studio

The Problem

If you look back at the screenshot, you will notice that the labeled `EditText` widgets are ragged, in terms of their layout. Each `EditText` immediately follows the `TextView` label, without regard to any sort of "columns". That is because each `LinearLayout` is largely independent. You cannot readily have one row depend upon the other rows.

A `TableLayout` would be a better choice for this sort of a form, as there we can have distinct columns of labels and fields. We will see how `TableLayout` handles this structure later in this chapter.

All Things Are Relative

`RelativeLayout`, as the name suggests, lays out widgets based upon their relationship to other widgets in the container and the parent container. You can place Widget X below and to the left of Widget Y, or have Widget Z's bottom edge align with the bottom of the container, and so on.

Concepts and Attributes

To make all this work, we need ways to reference other widgets within an XML layout file, plus ways to indicate the relative positions of those widgets.

Positions Relative to Container

The easiest relations to set up are tying a widget's position to that of its container:

1. `android:layout_alignParentTop` says the widget's top should align with the top of the container
2. `android:layout_alignParentBottom` says the widget's bottom should align with the bottom of the container
3. `android:layout_alignParentStart` says the widget's start side should align with the start side of the container
4. `android:layout_alignParentEnd` says the widget's end side should align with the end side of the container
5. `android:layout_centerHorizontal` says the widget should be positioned horizontally at the center of the container
6. `android:layout_centerVertical` says the widget should be positioned vertically at the center of the container
7. `android:layout_centerInParent` says the widget should be positioned both horizontally and vertically at the center of the container

All of these attributes take a simple boolean value (`true` or `false`). Also, there are `android:layout_alignParentLeft` and `android:layout_alignParentRight` attributes, for pre-Android 4.2 devices or for cases where you want to position irrespective of language direction.

Note that the padding of the widget is taken into account when performing these various alignments. The alignments are based on the widget's overall cell (combination of its natural space plus the padding).

Relative Notation in Attributes

The remaining attributes of relevance to `RelativeLayout` take as a value the identity of a widget in the container. To do this:

- Put identifiers (`android:id` attributes) on all elements that you will need to address

- Address these widgets from other widgets using the identifiers

The first occurrence of an id value should have the plus sign (`@+id/widget_a`); the second and subsequent times that id value is used in the layout file should drop the plus sign (`@id/widget_a`). This allows the build tools to better help you catch typos in your widget `id` values — if you do not have a plus sign for a widget `id` value that has not been seen before, that will be caught at compile time. For example, if Widget A appears in the `RelativeLayout` before Widget B, and Widget A is identified as `@+id/widget_a`, Widget B can refer to Widget A in one of its own attributes via the identifier `@id/widget_a`.

Positions Relative to Other Widgets

There are four attributes that control position of a widget *vis-à-vis* other widgets:

1. `android:layout_above` indicates that the widget should be placed above the widget referenced in the property
2. `android:layout_below` indicates that the widget should be placed below the widget referenced in the property
3. `android:layout_toStartOf` indicates that the widget should be placed to the start of the widget referenced in the property
4. `android:layout_toEndOf` indicates that the widget should be placed to the end of the widget referenced in the property

There are also `android:layout_toLeftOf` and `android:layout_toRightOf` attributes for use with older devices.

Beyond those four, there are five additional attributes that can control one widget's alignment relative to another:

1. `android:layout_alignTop` indicates that the widget's top should be aligned with the top of the widget referenced in the property
2. `android:layout_alignBottom` indicates that the widget's bottom should be aligned with the bottom of the widget referenced in the property
3. `android:layout_alignStart` indicates that the widget's starting edge should be aligned with the starting edge of the widget referenced in the property
4. `android:layout_alignEnd` indicates that the widget's ending edge should be aligned with the ending edge of the widget referenced in the property
5. `android:layout_alignBaseline` indicates that the baselines of the two widgets should be aligned (where the "baseline" is that invisible line that text appears to sit on)

The last one is useful for aligning labels and fields so that the text appears "natural". Since fields have a box around them and labels do not, `android:layout_alignTop` would align the top of the field's box with the top of the label, which will cause the text of the label to be higher on-screen than the text entered into the field.

Android Studio Graphical Layout Editor

You will find `RelativeLayout` in the "Layouts" section of the Palette in the Android Studio Graphical Layout editor. You can drag that into your layout XML resource.

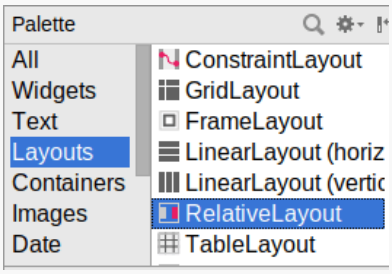


Figure 129: Layouts Section of Palette, RelativeLayout Highlighted

As you drag other widgets into your `RelativeLayout`, you will see arrows hinting at the rules that will be applied if you drop the widget at the current mouse location:

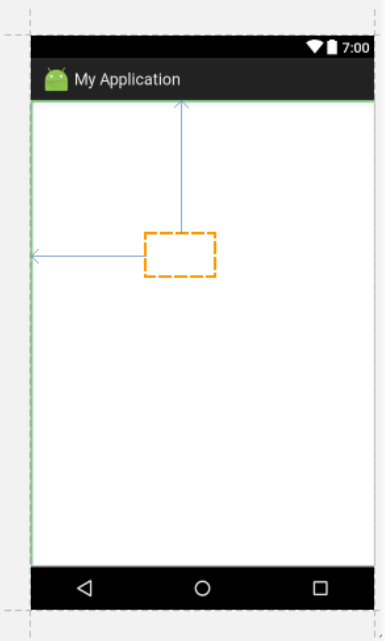


Figure 130: Dragging a Widget in a RelativeLayout

Getting the rules that you *want* may or may not be possible purely through drag-and-drop. You may need to just drop the widget into the `RelativeLayout` and manually adjust the rules, whether by using the Attributes pane or by editing the XML directly.

Example: Bottom-then-Top

Earlier in the chapter, we saw how to implement the bottom-then-top pattern using a `LinearLayout`, where we had a small button on the bottom and a large button on the top. The large button was set to take up all space that was not required by the small button.

We can achieve the same result using a `RelativeLayout`.

The XML

As with the `LinearLayout` scenario, we have one container plus the two `Button` widgets. In this case, the container is a `RelativeLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_above="@+id/another_button"
    android:layout_alignParentTop="true"
    android:text="@string/button" />

  <Button
    android:id="@id/another_button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:text="@string/another_button" />
</RelativeLayout>
```

(from Containers/Sampler/app/src/main/res/layout/bottom_then_top_rl.xml)

The bottom `Button` has the same size as before, with `android:layout_width` set to `match_parent` and `android:layout_height` set to `wrap_content`. However, it also has `android:layout_alignParentBottom="true"`, anchoring it to the bottom of the `RelativeLayout`. Since the `RelativeLayout` fills all available space, the bottom `Button` is anchored to the bottom of the screen, in effect.

The top `Button` has two `RelativeLayout` positioning attributes:

- `android:layout_above="@+id/another_button"`, so it is placed above the bottom `Button`
- `android:layout_alignParentTop="true"`, so it is anchored to the top of the `RelativeLayout`

Given these rules, it does not matter what the `android:layout_height` attribute value is. The `Button` will be stretched between those two anchor points: the top of the `RelativeLayout` and the top of the bottom `Button`.

Android Studio Graphical Layout Editor

By default, when creating a new layout resource file, you get either a `ConstraintLayout` or a `vertical LinearLayout` as the root element. You can change that by replacing the "Root element" value with any other widget or container class name, such as `RelativeLayout`:

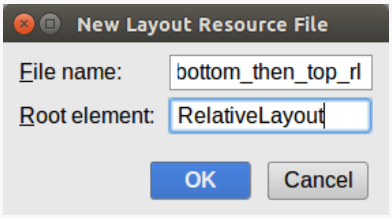


Figure 131: New Layout Resource with RelativeLayout

However, from there, using the drag-and-drop capabilities will start to be more of a pain than they are worth.

You can drag a `Button` into the `RelativeLayout`, for example:

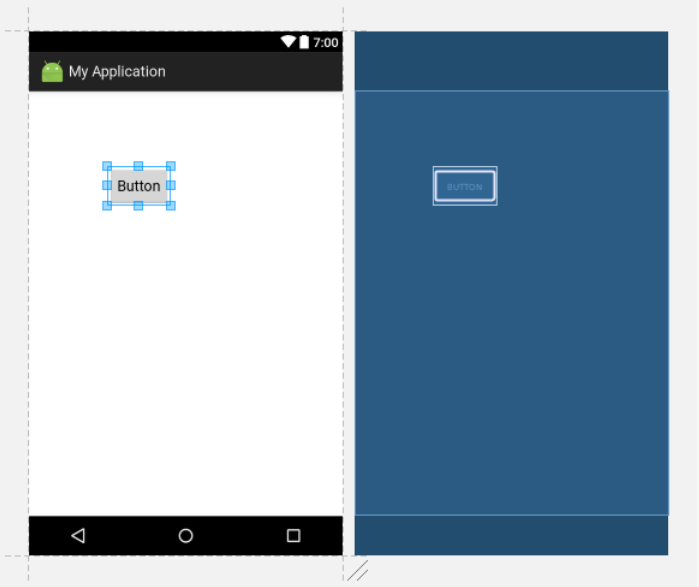


Figure 132: RelativeLayout and Button

However, you get no visual feedback after you drop the widget of what the rules are that the IDE chose, based on your drag-and-drop location. If you rummage through the Attributes pane, you will see that the rules are reflected in some checkbox attributes for the boolean `RelativeLayout` `android:layout_` attributes, plus margins based on how far from the `RelativeLayout` edges you placed the widget:

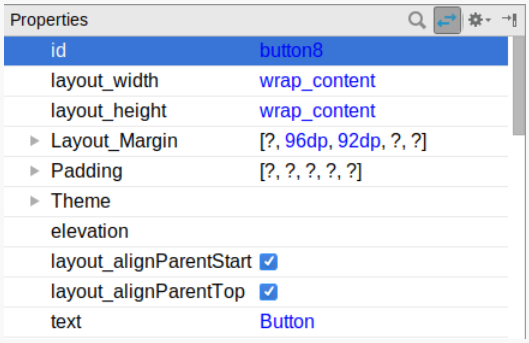


Figure 133: Button Attributes in RelativeLayout

Fixing those through the Attributes pane is no easier than is fixing them through the XML editor. Arguably, using the Attributes pane is slower.

Example: URL Dialog

We also used `LinearLayout` to create the "URL dialog" UI, where we had the labeled field along with "OK" and "Cancel" buttons. That same structure can also be built using a `RelativeLayout`.

With `LinearLayout`, we needed three containers: one `vertical` `LinearLayout` wrapped around two `horizontal` `LinearLayouts`. `RelativeLayout` is simpler from that standpoint, as we only need one `RelativeLayout`. The complexity moves into the widgets instead:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_marginLeft="4dp"
    android:layout_marginStart="4dp"
    android:text="@string/url"/>

  <EditText
```

```
    android:id="@id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_toRightOf="@id/label"
    android:layout_toEndOf="@id/label"
    android:inputType="text"/>

  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@id/entry"
    android:layout_alignEnd="@id/entry"
    android:layout_below="@id/entry"
    android:text="@string/ok"/>

  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/ok"
    android:layout_toLeftOf="@id/ok"
    android:text="@string/cancel"/>

</RelativeLayout>
```

(from Containers/Sampler/app/src/main/res/layout/url_dialog_rl.xml)

All the `RelativeLayout` itself needs is its size, set the same as with `vertical LinearLayout` in the earlier version of this sample — a width of `match_parent` and a height of `wrap_content`.

Other than `android:layout_width` and `android:layout_height`, all of the widget attributes with `layout_` are rules used by children of a `RelativeLayout` for positioning. So, we have:

Widget	Horizontal Anchor	Vertical Anchor
`TextView`	left/start side of the `RelativeLayout`, with `4dp` of margin	baseline of the `EditText`
`EditText`	right/end side of the `TextView`	top of the `RelativeLayout`
“OK” `Button`	right/end edge of the `EditText`	bottom of the `EditText`

Widget	Horizontal Anchor	Vertical Anchor
"Cancel" `Button`	left/start edge of the "OK" `Button`	top of the "OK" `Button`

Since the `EditText` width is set to `match_parent`, it will fill all the space in the "row" after the `TextView`. Since the "OK" `Button` horizontal position is tied to the `EditText`, the `Button` slides over to the edge of the screen, dragging along the connected "Cancel" `Button`.

As a result, we get the same basic UI:

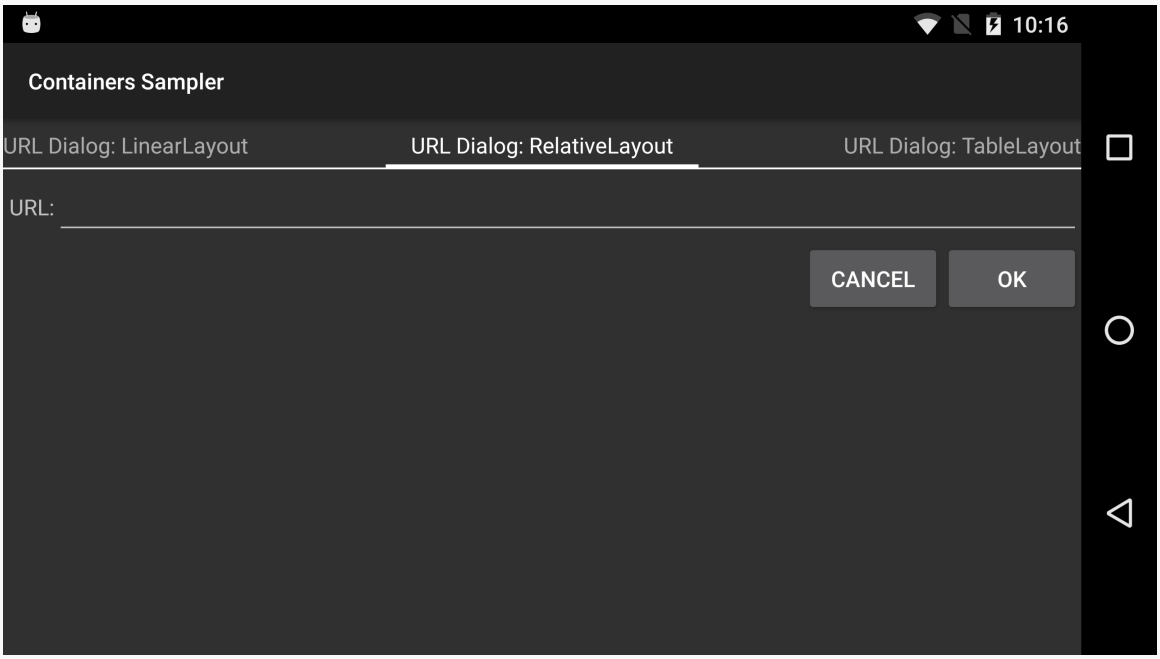


Figure 134: URL-Dialog Layout, Using RelativeLayout

Example: Overlap

`RelativeLayout` also has a feature that `LinearLayout` lacks — the ability to have widgets overlap one another. Later children of a `RelativeLayout` are "higher in the Z axis" than are earlier children, meaning that later children will overlap earlier children if they are set up to occupy the same space in the layout.

Here, we have two buttons, where the "I am small" button overlaps the "I am big" button:

Figure 135: Overlap Layout, Using RelativeLayout

The layout is fairly simple:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/big"
    android:textSize="120dip"
    android:textStyle="bold"/>

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="@string/small"/>

</RelativeLayout>
```

(from Containers/Sampler/app/src/main/res/layout/overlap_rl.xml)

The first `Button` is set to fill the screen. The second `Button` is set to be centered inside the parent, but only take up as much space as is needed for its caption. Hence, the second `Button` will appear to "float" over the first `Button`.

Both `Button` widgets can still be clicked, though clicking on the smaller `Button` does not also click the bigger `Button`. Your clicks will be handled by the widget on top in the case of an overlap like this.

On Android 5.0 and higher, it is possible to achieve a similar effect with `LinearLayout` by using the `android:elevation` attribute to control the Z axis, where higher elevation values mean higher on the Z axis, floating over those that are lower on the Z axis.

Tabula Rasa

If you like HTML tables, you will like Android's `TableLayout`. It allows you to position your widgets in a grid to your specifications. You control the number of rows and columns, which columns might shrink or stretch to accommodate their contents, and so on.

`TableLayout` works in conjunction with `TableRow`. `TableLayout` controls the overall behavior of the container, with the widgets themselves poured into one or more `TableRow` containers, one per row in the grid.

Concepts and Attributes

For all this to work, we need to figure out how widgets work with rows and columns, plus how to handle widgets that live outside of rows.

Putting Cells in Rows

Rows are declared by you, the developer, by putting widgets as children of a `TableRow` inside the overall `TableLayout`. You, therefore, control directly how many rows appear in the table.

The number of columns are determined by Android; you control the number of columns in an indirect fashion.

First, there will be at least one column per widget in your longest row. So if you have three rows, one with two widgets, one with three widgets, and one with four widgets, there will be at least four columns.

However, a widget can take up more than one column by including the `android:layout_span` property, indicating the number of columns the widget spans. This is akin to the `colspan` attribute one finds in table cells in HTML:

```
<TableRow>
  <TextView android:text="URL:" />
  <EditText
    android:id="@+id/entry"
    android:layout_span="3"/>
</TableRow>
```

In the above XML layout fragment, the field spans three columns.

Ordinarily, widgets are put into the first available column. In the above fragment, the label would go in the first column (column `0`, as columns are counted starting from `0`), and the field would go into a spanned set of three columns (columns `1` through `3`). However, you can put a widget into a different column via the `android:layout_column` property, specifying the `0`-based column the widget belongs to:

```
<TableRow>
  <Button
    android:id="@+id/cancel"
    android:layout_column="2"
    android:text="Cancel" />
  <Button android:id="@+id/ok" android:text="OK" />
</TableRow>
```

In the preceding XML layout fragment, the Cancel button goes in the third column (column `2`). The OK button then goes into the next available column, which is the fourth column.

Non-Row Children of TableLayout

Normally, `TableLayout` contains only `TableRow` elements as immediate children. However, it is possible to put other widgets in between rows. For those widgets, `TableLayout` behaves a bit like `LinearLayout` with vertical orientation. The widgets automatically have their width set to `match_parent`, so they will fill the same space that the longest row does.

Stretch, Shrink, and Collapse

By default, each column will be sized according to the "natural" size of the widest widget in that column (taking spanned columns into account). Sometimes, though, that does not work out very well, and you need more control over column behavior.

You can place an `android:stretchColumns` property on the `TableLayout`. This lists the column or columns that should absorb any extra space on the row, if the natural width of the columns collectively is narrower than the available horizontal space. You can:

- List a single column to be stretched (e.g., `android:stretchColumns="0"` to stretch the first column)
- Provide a comma-delimited list of columns to be stretched (e.g., `android:stretchColumns="0,1"` to stretch the first two columns)
- Use `*` to indicate that all columns should be stretched, akin to using equal `android:layout_weight` values in a horizontal `LinearLayout` (e.g., `android:stretchColumns="*"`)

Conversely, you can place an `android:shrinkColumns` property on the `TableLayout`. Again, this should be a single column number, a comma-delimited list of column numbers, or `*` as shorthand for referring to all columns. The columns listed in this property will try to word-wrap their contents to reduce the effective width of the column — by default, widgets are not word-wrapped. This helps if you have columns with potentially wordy content that might cause some columns to be pushed off the right side of the screen.

You can also leverage an `android:collapseColumns` property on the `TableLayout`, again with a column number or comma-delimited list of column numbers (`*` is not documented as an available option). These columns will start out "collapsed", meaning they will be part of the table information but will be invisible. Programmatically, you can collapse and un-collapse columns by calling `setColumnCollapsed()` on the `TableLayout`. You might use this to allow users to control which columns are of importance to them and should be shown versus which ones are less important and can be hidden.

You can also control stretching and shrinking at runtime via `setColumnStretchable()` and `setColumnShrinkable()`.

Android Studio Graphical Layout Editor

You will find `TableLayout` and `TableRow` in the "Layouts" section of the Palette in the Android Studio graphical layout editor:

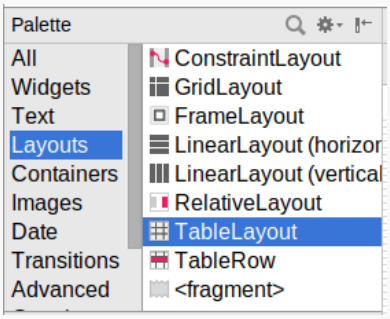


Figure 136: Layouts Section of Palette, TableLayout Highlighted

Given a `TableLayout`, you can drag one or more `TableRow` containers into it, then start dragging widgets into the rows, much as you might set up nested `LinearLayout` containers.

Example: A Bigger Form

One area where `TableLayout` excels is with forms, particularly if you are using the classic two-column "label and widget" structure for the form. This is because `TableLayout` can give you real columns, whereas `LinearLayout` and `RelativeLayout` cannot.

As with most `TableLayout` usages, the immediate children of ours are mostly `TableRow` containers, each providing the contents for the columns:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="8dp"
  android:stretchColumns="1">

  <TableRow>
    <TextView android:text="@string/name" />
    <EditText android:inputType="text" />
  </TableRow>

  <TableRow>
```

```
    <TextView android:text="@string/home_planet" />
    <EditText android:inputType="text" />
  </TableRow>

  <TableRow>
    <CheckBox
      android:layout_column="1"
      android:text="@string/android_programmer" />
  </TableRow>

  <TableRow>
    <TextView android:text="@string/favorite_food" />
    <EditText android:inputType="text" />
  </TableRow>

  <Button android:text="@string/do_something" />

</TableLayout>
```

(from Containers/Sampler/app/src/main/res/layout/form_tl.xml)

The `TableLayout` itself has `android:stretchColumns="1"`, so all leftover space in the rows will go to the second column (with the first column having an index of 0).

The first, second, and fourth `TableRow` each have the same structure, with a `TextView` label preceding the `EditText` where the user can fill in the data. The third `TableRow`, though, has only one child: the `CheckBox`. And, our `Button` lies outside of any `TableRow`, as a direct child of the root `TableLayout`. Both the `CheckBox` and the `Button` will exist on a row of their own. The difference is that the `CheckBox` goes in the second column, courtesy of `android:layout_column="1"`, whereas the `Button` will span the entire row (the way `TableRow` containers span the entire width of the `TableLayout`).

So, compared with the original `LinearLayout` version of this sample, our `TableLayout` columns are neat and aligned:

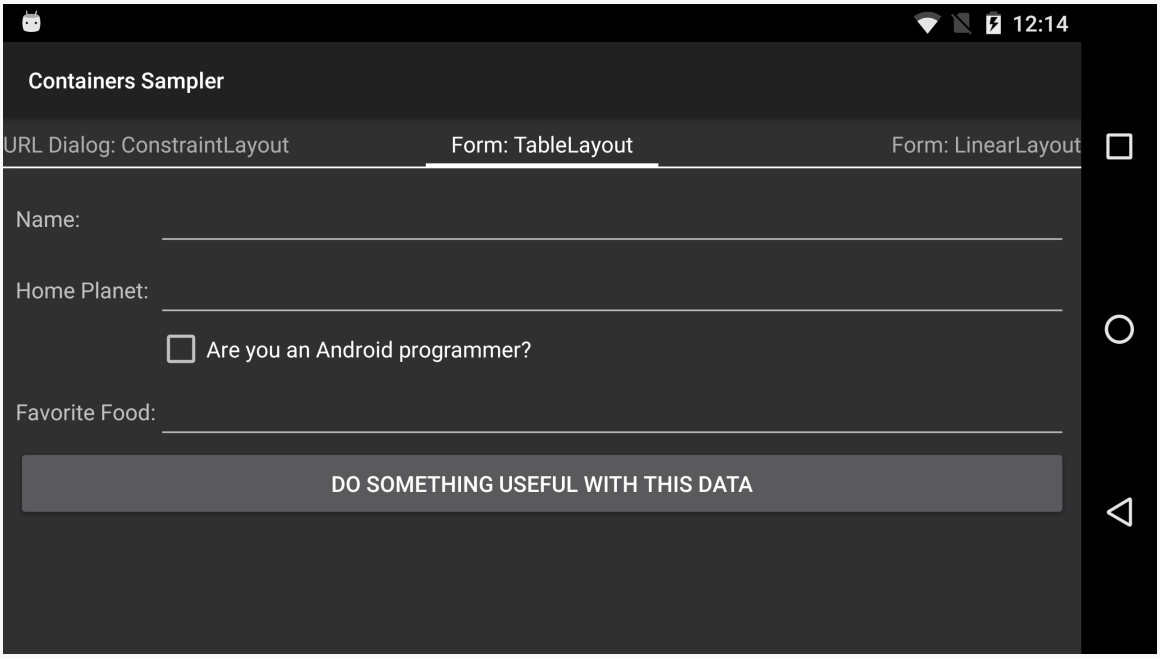


Figure 137: Form Layout, Using TableLayout

Example: URL Dialog

The "URL dialog" layout, previously seen implemented using `LinearLayout` and `RelativeLayout`, can also be implemented using a `TableLayout`. This is not the most natural use of a `TableLayout`, but you can do it if you wanted.

As with the form sample above, we start with a root `TableLayout` having `android:stretchColumns="1"` to give all extra space to the second column... even though we will wind up with a total of four columns this time:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:stretchColumns="1">

  <TableRow>

    <TextView
      android:layout_marginLeft="4dip"
      android:layout_marginStart="4dip"
      android:text="@string/url" />
```

```
    <EditText
      android:id="@+id/entry"
      android:layout_span="3"
      android:inputType="text" />
  </TableRow>

  <TableRow>

    <Button
      android:id="@+id/cancel"
      android:layout_column="2"
      android:text="@string/cancel" />

    <Button
      android:id="@+id/ok"
      android:text="@string/ok" />
  </TableRow>

</TableLayout>
```

(from Containers/Sampler/app/src/main/res/layout/url_dialog_tl.xml)

The `EditText` in the first `TableRow` has `android:layout_span="3"`, indicating that it should span to fill three columns. That, plus our one `TextView`, means that the first row is set up for four columns in total.

The first `Button` in the second `TableRow` has `android:layout_column="2"`, indicating that it should go into the third column. The other `Button` will go into the next column (the fourth column in this case), and the first two columns are skipped. So, this row also is set up for four columns.

So, when `android:stretchColumns="1"` is applied, the extra space will be given to the "contents" of the second column:

- the `EditText` in the first row
- the empty space preceding the two `Button` widgets in the second row

Hey, What About ConstraintLayout?

In 2016, Google introduced `ConstraintLayout`, with a vision of it becoming the fourth major container and perhaps the default one that you would choose. `ConstraintLayout` has its benefits, to be certain. However, it requires the use of a library, and we have not yet covered how to attach libraries to an Android module.

So, we will discuss `ConstraintLayout` a bit later in the book.

Turning Back to RTL

In order for the "start"/"end" attributes to work, you need to have `android:supportsRtl="true"` in your `<application>` element in your manifest. Most newly-created projects will have this attribute already set for you by the new-project wizard.

To see how your app behaves with RTL — without having to learn Arabic or Hebrew, if you are not literate in those languages — you can force Android to use RTL layout rules with any language on Android 4.2+ devices. To do this, go into the Settings app of the device or emulator and choose "Developer options". In there, scroll down to the "Force RTL layout direction" item. By default, this is turned off, and so layout direction is determined by the user's chosen language:

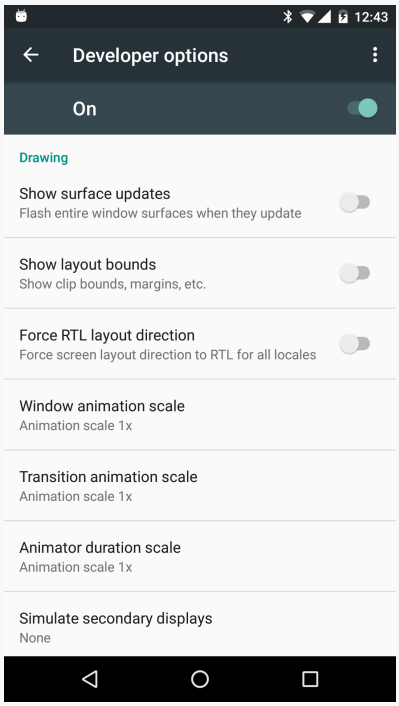


Figure 138: Developer Options in Settings, Normal Mode

Tapping that switch uses RTL layout rules — with "start" referring to the right and "end" referring to the left — for all languages:

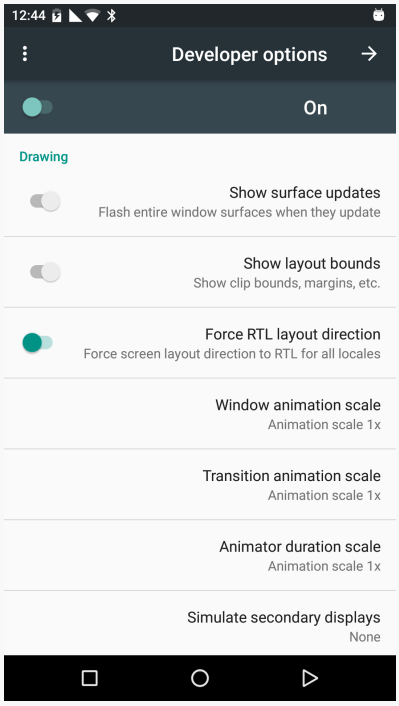


Figure 139: Developer Options in Settings, Forced-RTL Mode

As a reminder, if "Developer options" is not in the list of Settings categories, go into the "About device" category, find the build number item, and tap on it seven times. This will enable "Developer options" back on the main list of Settings categories.

Other Common Widgets and Containers

In the chapter on basic widgets, we left out all of the classic "two-state" widgets, such as checkboxes and radio buttons. We will examine those and other related widgets in this chapter.

Beyond the classic general-purpose containers (`LinearLayout`, `RelativeLayout`, `TableLayout`), there are other specialized containers, like `FrameLayout` and `RadioGroup`, that you will use from time to time. We will examine those in this chapter as well.

Just a Box to Check

The classic checkbox has two states: checked and unchecked. Clicking the checkbox toggles between those states to indicate a choice (e.g., "Add rush delivery to my order").

In Android, there is a `CheckBox` widget to meet this need. It has `TextView` as an ancestor, so you can use `TextView` properties like `android:textColor` to format the widget.

Within Java, you can invoke:

1. `isChecked()` to determine if the checkbox has been checked
2. `setChecked()` to force the checkbox into a checked or unchecked state
3. `toggle()` to toggle the checkbox as if the user clicked upon it

Also, you can register a listener object (in this case, an instance of `OnCheckedChangeListener`) to be notified when the state of the checkbox changes.

For example, from the `Basic/CheckBox` sample project, here is a simple checkbox layout:

```
<?xml version="1.0" encoding="utf-8"?>
<CheckBox xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/check"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/unchecked"/>
```

(from Basic/CheckBox/app/src/main/res/layout/main.xml)

The corresponding `CheckBoxDemo.java` retrieves and configures the behavior of the checkbox:

```
package com.commonsware.android.checkbox;

import android.app.Activity;
import android.os.Bundle;
import android.widget.CheckBox;
import android.widget.CompoundButton;

public class CheckBoxDemo extends Activity implements
    CompoundButton.OnCheckedChangeListener {
  CheckBox cb;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    cb=(CheckBox)findViewById(R.id.check);
    cb.setOnCheckedChangeListener(this);
  }

  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    if (isChecked) {
      cb.setText(R.string.checked);
    }
    else {
      cb.setText(R.string.unchecked);
    }
```

```
  }
}
```

(from Basic/CheckBox/app/src/main/java/com/commonsware/android/checkbox/CheckBoxDemo.java)

Note that the activity serves as its own listener for checkbox state changes since it implements the `OnCheckedChangeListener` interface (set via `cb.setOnCheckedChangeListener(this)`). The callback for the listener is `onCheckedChanged()`, which receives the checkbox whose state has changed and what the new state is. In this case, we update the text of the checkbox to reflect what the actual box contains.

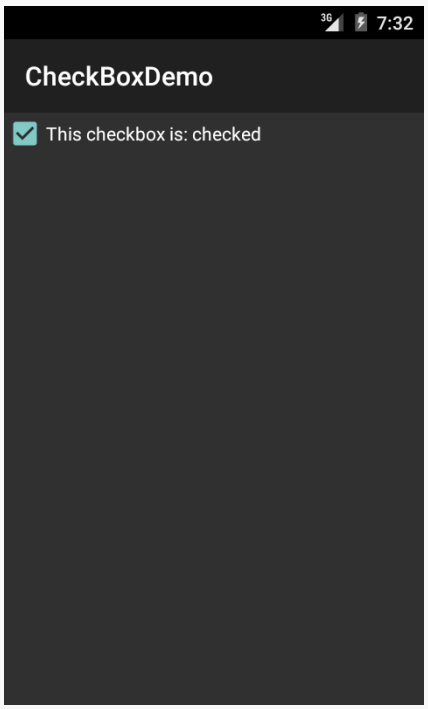


Figure 140: CheckBoxDemo Sample App, with CheckBox Checked

Android Studio Graphical Layout Editor

The `CheckBox` widget can be found in the "Widgets" portion of the Palette in the Android Studio Graphical Layout editor:

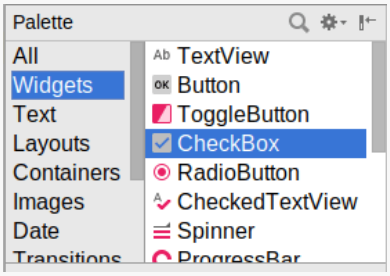


Figure 141: Widgets Palette, CheckBox Shown Highlighted

You can drag it into the layout and configure it as desired using the Attributes pane. As `CheckBox` inherits from `TextView`, most of the settings are the same as those you would find on a regular `TextView`.

Don't Like Checkboxes? How About Toggles or Switches?

A similar widget to `CheckBox` is `ToggleButton`. Like `CheckBox`, `ToggleButton` is a two-state widget that is either checked or unchecked. However, `ToggleButton` has a distinct visual appearance:

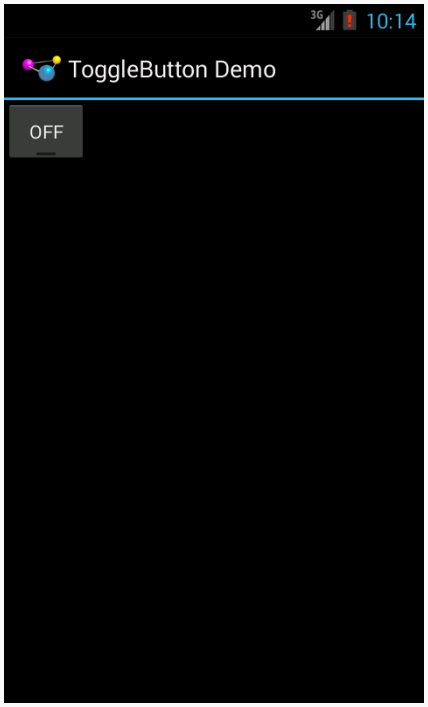


Figure 142: ToggleButtonDemo Sample, Unchecked

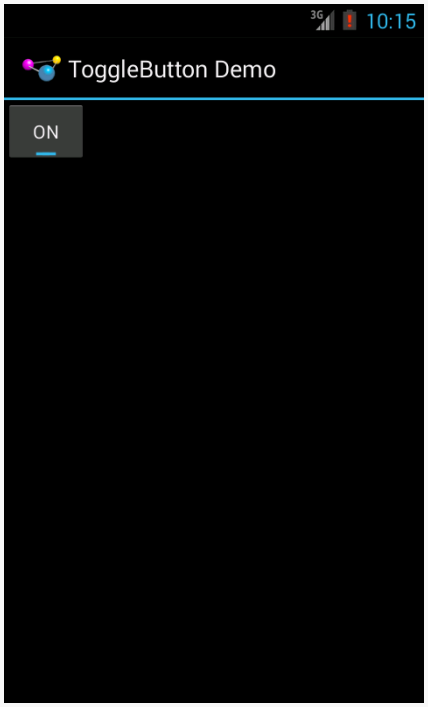


Figure 143: ToggleButtonDemo Sample, Checked

Otherwise, `ToggleButton` behaves much like `CheckBox`. You can put it in a layout file, as seen in the `Basic/ToggleButton` sample:

```
<?xml version="1.0" encoding="utf-8"?>
<ToggleButton xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

(from Basic/ToggleButton/app/src/main/res/layout/main.xml)

You can also set up an `OnCheckedChangeListener` to be notified when the user changes the state of the `ToggleButton`.

Similarly, Android has a `Switch` widget, showing the state via a small "ON/OFF" slider:

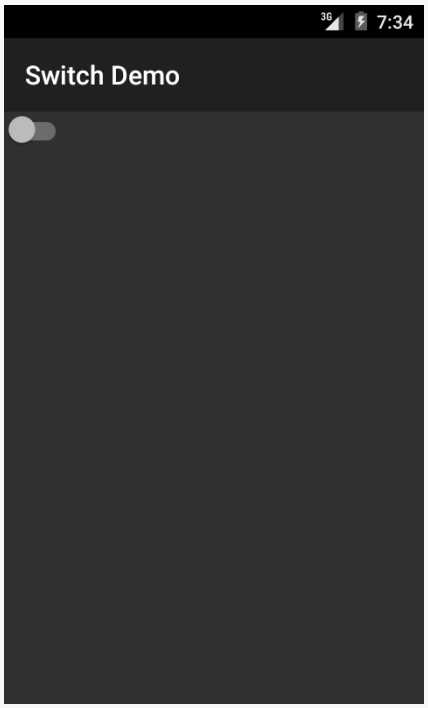


Figure 144: SwitchDemo Sample, Unchecked

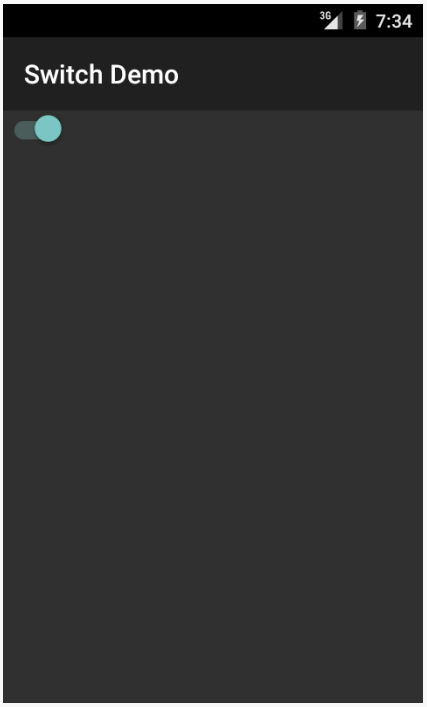


Figure 145: SwitchDemo Sample, Checked

`Switch`, like `CheckBox` and `ToggleButton`, inherits from `CompoundButton`, and therefore shares a common API, for methods like `toggle()`, `isChecked()`, and `setChecked()`. And, as with the others, you can put it in a layout file, as seen in the `Basic/Switch` sample:

```
<?xml version="1.0" encoding="utf-8"?>
<Switch xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

(from Basic/Switch/app/src/main/res/layout/main.xml)

The biggest limitation with `Switch` is that it was only added to the Android SDK in API Level 14. If your `minSdkVersion` is set to 14 or higher, you are welcome to use `Switch`. If your `minSdkVersion` is set to something lower than 14, though, you will either need to choose something else or get into more complicated scenarios, like using a library that offers a backport of `Switch`. We will cover those more complicated scenarios later in the book; for now, it is simplest to only use `Switch` if your `minSdkVersion` is set to 14 or higher.

Android Studio Graphical Layout Editor

The `ToggleButton` and `Switch` widgets can be found in the "Widgets" portion of the Palette in the Android Studio Graphical Layout editor, just beneath the `CheckBox` widget:

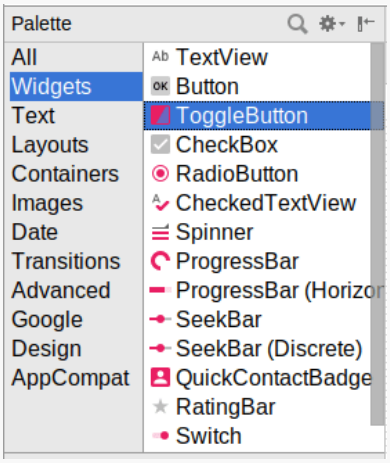


Figure 146: Widgets Palette, ToggleButton Highlighted and Switch At Bottom

You can drag either widget into the layout and configure it as desired using the Attributes pane.

Turn the Radio Up

As with other implementations of radio buttons in other toolkits, Android's radio buttons are two-state, like checkboxes, but can be grouped such that only one radio button in the group can be checked at any time.

`CheckBox`, `ToggleButton`, `Switch`, and `RadioButton` all inherit from `CompoundButton`, which in turn inherits from `TextView`. Hence, all the standard `TextView` properties for font face, style, color, etc. are available for controlling the look of radio buttons. Similarly, you can call `isChecked()` on a `RadioButton` to see if it is selected, `toggle()` to change its checked state, and so on, like you can with a `CheckBox`.

Most times, you will want to put your `RadioButton` widgets inside of a `RadioGroup`. The `RadioGroup` is a `LinearLayout` that indicates a set of radio buttons whose state is tied, meaning only one button out of the group can be selected at any time. If you assign an `android:id` to your `RadioGroup` in your XML layout, you can access the group from your Java code and invoke:

1. `check()` to check a specific radio button via its ID (e.g., `group.check(R.id.radio1)`)
2. `clearCheck()` to clear all radio buttons, so none in the group are checked
3. `getCheckedRadioButtonId()` to get the ID of the currently-checked radio button (or `-1` if none are checked)

Note that the mutual-exclusion feature of `RadioGroup` only applies to `RadioButton` widgets that are immediate children of the `RadioGroup`. You cannot have other containers between the `RadioGroup` and its `RadioButton` widgets.

For example, from the `Basic/RadioButton` sample application, here is an XML layout showing a `RadioGroup` wrapping a set of `RadioButton` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
    <RadioButton android:id="@+id/radio1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/rock" />

    <RadioButton android:id="@+id/radio2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/scissors" />

    <RadioButton android:id="@+id/radio3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/paper" />
</RadioGroup>
```

(from Basic/RadioButton/app/src/main/res/layout/main.xml)

Using the stock Android-generated Java for the project and this layout, you get:

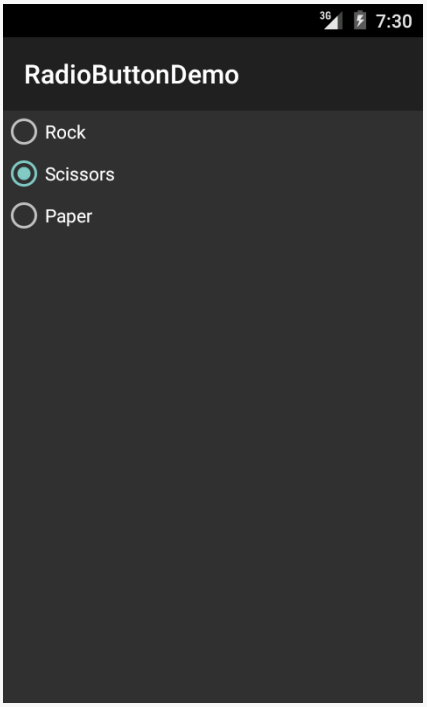


Figure 147: RadioButtonDemo, with "Scissors" Checked

Note that the radio button group is initially set to be completely unchecked at the outset. To preset one of the radio buttons to be checked, use either `setChecked()` on the `RadioButton` or `check()` on the `RadioGroup` from within your `onCreate()` callback in your activity. Alternatively, you can use the `android:checked` attribute on one of the `RadioButton` widgets in the layout file.

Android Studio Graphical Layout Editor

The `RadioGroup` container can be found in the "Containers" portion of the Palette in the Android Studio Graphical Layout editor:

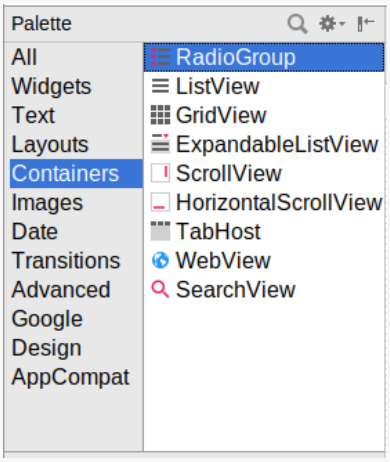


Figure 148: Widgets Palette, RadioGroup Highlighted

Dragging a `RadioGroup` into the preview works much like dragging a `LinearLayout` into the preview. You get a box into which you can drag other widgets, such as the `RadioButton` found in the "Widgets" section of the Palette.

Scrollwork

Phone screens tend to be small, which requires developers to use some tricks to present a lot of information in the limited available space. One trick for doing this is to use scrolling, so only part of the information is visible at one time, the rest available via scrolling up or down.

`ScrollView` is a container that provides scrolling for its contents. You can take a layout that might be too big for some screens, wrap it in a `ScrollView`, and still use your existing layout logic. It just so happens that the user can only see part of your layout at one time, the rest available via scrolling.

For example, here is a `ScrollView` used in an XML layout file (from the `Containers/Scroll` demo):

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    android:stretchColumns="0">
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#000000"/>
      <TextView android:text="#000000"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#440000" />
      <TextView android:text="#440000"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#884400" />
      <TextView android:text="#884400"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#aa8844" />
      <TextView android:text="#aa8844"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#ffaa88" />
      <TextView android:text="#ffaa88"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#ffffaa" />
      <TextView android:text="#ffffaa"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
```

```
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#ffffff" />
      <TextView android:text="#ffffff"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
  </TableLayout>
</ScrollView>
```

(from Containers/Scroll/app/src/main/res/layout/main.xml)

Without the `ScrollView`, the table would take up at least 560 density-independent pixels (7 rows at 80 dips each, based on the `View` declarations). There may be some devices with screens capable of showing that much information, but many will be smaller. The `ScrollView` lets us keep the table as-is, but only present part of it at a time.

On the stock Android emulator, when the activity is first viewed, you see:

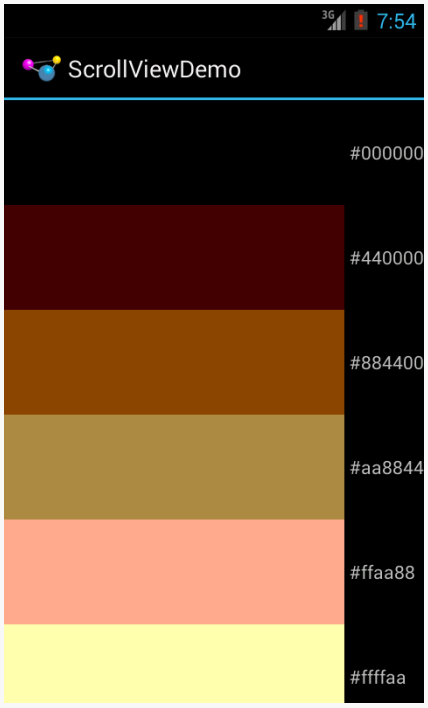


Figure 149: The ScrollViewDemo sample application

Notice how only five rows and part of the sixth are visible. You can scroll up and down to see the remaining rows. Also note how the right side of the content gets clipped by the scrollbar — be sure to put some padding on that side or otherwise ensure your own content does not get clipped in that fashion.

Android also has `HorizontalScrollView`, which works like `ScrollView`... just horizontally. This would be good for forms that might be too wide rather than too tall. Note that `ScrollView` only scrolls vertically and `HorizontalScrollView` only scrolls horizontally.

Also, note that you cannot put scrollable items into a `ScrollView`. For example, a `ListView` widget — which we will see in an upcoming chapter — already knows how to scroll. You do not need to put a `ListView` in a `ScrollView`, and if you were to try, it would not work very well.

And, a `ScrollView` or `HorizontalScrollView` can only have one child — if you want more than one, wrap the children in a suitable container class (e.g., a `LinearLayout`) and put that inside the `ScrollView` or `HorizontalScrollView`.

Android Studio Graphical Layout Editor

The `ScrollView` and `HorizontalScrollView` widgets appear in the "Containers" section of the Palette in the Graphical Layout editor. You can drag one of these into your layout XML resource, then drag *one* child into it.

Making Progress with ProgressBars

If you are going to fork background threads to do work on behalf of the user, you will want to think about keeping the user informed that work is going on. This is particularly true if the user is effectively waiting for that background work to complete.

The typical approach to keeping users informed of progress is some form of progress bar, like you see when you copy a bunch of files from place to place in many desktop operating systems. Android supports this through the `ProgressBar` widget.

A `ProgressBar` keeps track of progress, defined as an integer, with `0` indicating no progress has been made. You can define the maximum end of the range — what value indicates progress is complete — via `setMax()`. By default, a `ProgressBar` starts with a progress of `0`, though you can start from some other position via

`setProgress()`. Android 8.0 adds `setMin()` and `android:min` to set the lower end of the range to a custom value, instead of the default of `0`.

If you prefer your progress bar to be indeterminate — meaning that it will show a general animated effect, rather than a specific amount of progress – use `setIndeterminate()`, setting it to `true`.

In your Java code, you can either positively set the amount of progress that has been made (via `setProgress()`) or increment the progress from its current amount (via `incrementProgressBy()`). You can find out how much progress has been made via `getProgress()`.

Framing the Scene

Android has a `FrameLayout` class. Like `LinearLayout`, `RelativeLayout`, and `TableLayout`, `FrameLayout` exists to size and position its children. However, `FrameLayout` has a very simple pair of layout rules:

1. All children go in the upper-start corner (e.g., upper-left for LTR languages), unless `android:gravity` indicates to position the children elsewhere
2. Later children are higher on the Z axis than are earlier children, as with `RelativeLayout`, so children can overlap

The result is that all the widgets are stacked one on top of another.

This may seem useless. And, truth be told, it is not used nearly as commonly as are other containers.

Primarily, `FrameLayout` is used in places where we want to reserve space for something, but we do not know what the "something" is at compile time. The decision of what the "something" is will be made at runtime, where we will use Java code to put something in the `FrameLayout`. We will see this pattern used with fragments, later in the book.

Occasionally, `FrameLayout` is literally used for "framing", where we want some sort of a border around a child. In this case, the background of the `FrameLayout` (e.g., `android:background`) defines what the frame should look like. We will see this approach used in a few places, such as in the chapter on adding drag-and-drop to your app.

Visit the Trails!

The trails portion of the book contains a widget catalog, providing capsule descriptions and samples for a number of widgets not described elsewhere in this book.

You might also be interested in `GridLayout`, which is an alternative to the classic `LinearLayout`, `RelativeLayout`, and `TableLayout` containers.

Tutorial #5 - Creating a Layout

Later in these tutorials, we are going to allow the user to write down notes related to the book. These notes will be stored in a database and can be viewed, modified, or deleted as the user sees fit.

In this tutorial, we are going to set up the layout resource to allow the user to fill in these notes.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Creating a New Layout Resource

Right-click over the `res/layout/` directory and choose New > "Layout resource file" from the context menu. This brings up the New Layout Resource File dialog:

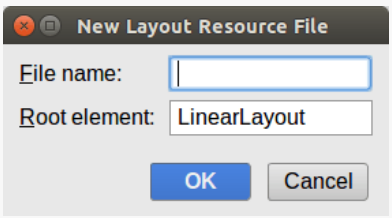


Figure 150: Android Studio New Layout Resource File Dialog

Fill in `editor` as the "Layout File Name", leave the rest of the dialog alone, and click the "OK" button.

Step #2: Defining the UI

That should have opened up the graphical layout editor for this new `editor` layout resource:

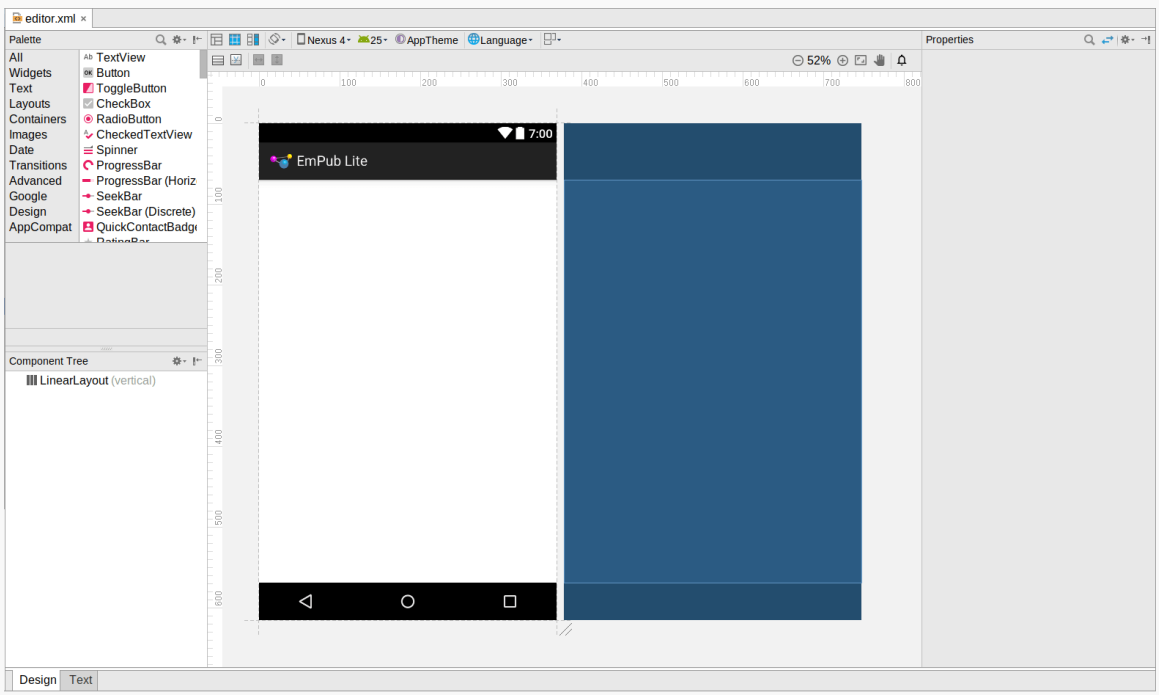


Figure 151: Graphical Layout Editor

Drag a "Multiline Text" widget from the Text section of the Palette into the preview area. In the attributes pane, change the `android:layout_width` and `android:layout_height` each to be `match_parent` and change the ID to `editor`:

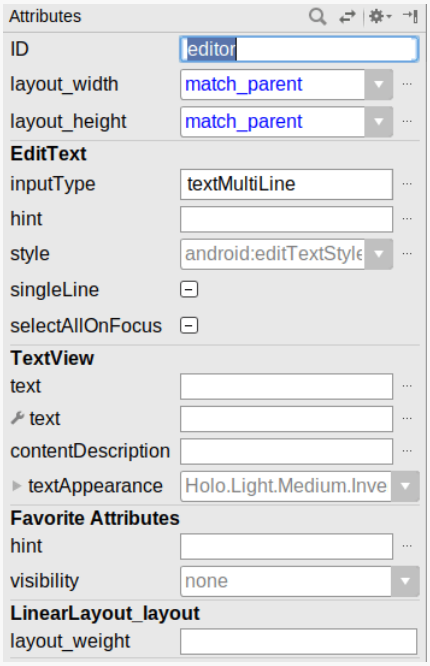


Figure 152: Graphical Layout Editor Attributes Pane

Next, in the attributes pane, click on the "hint" entry, then click the "..." button to the right of it. This will open up a string resource picker dialog:

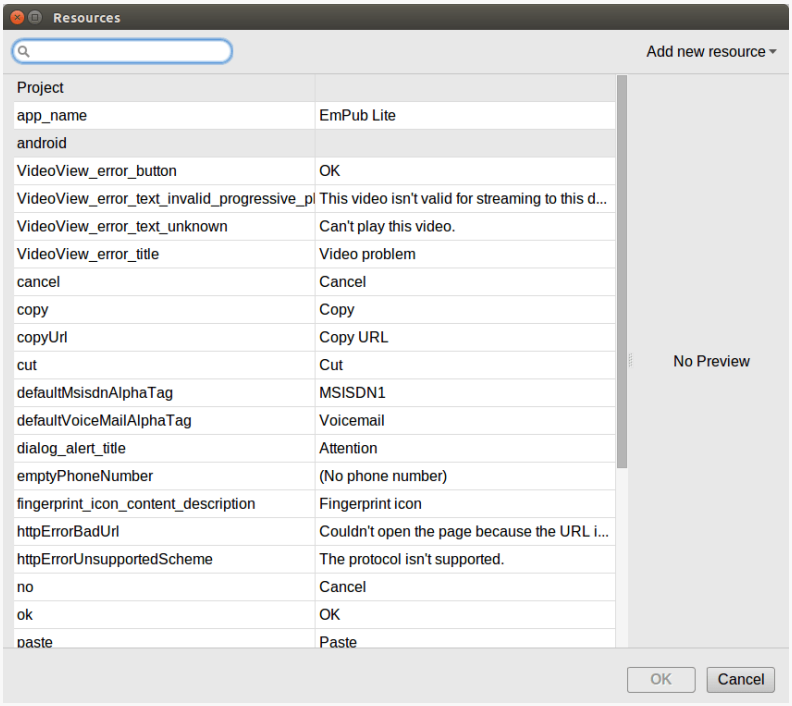


Figure 153: Android Studio String Resource Picker Dialog

Towards the upper right, click the "Add new resource" drop-down and choose "New string Value..." from it, to bring up the string resource editor dialog:

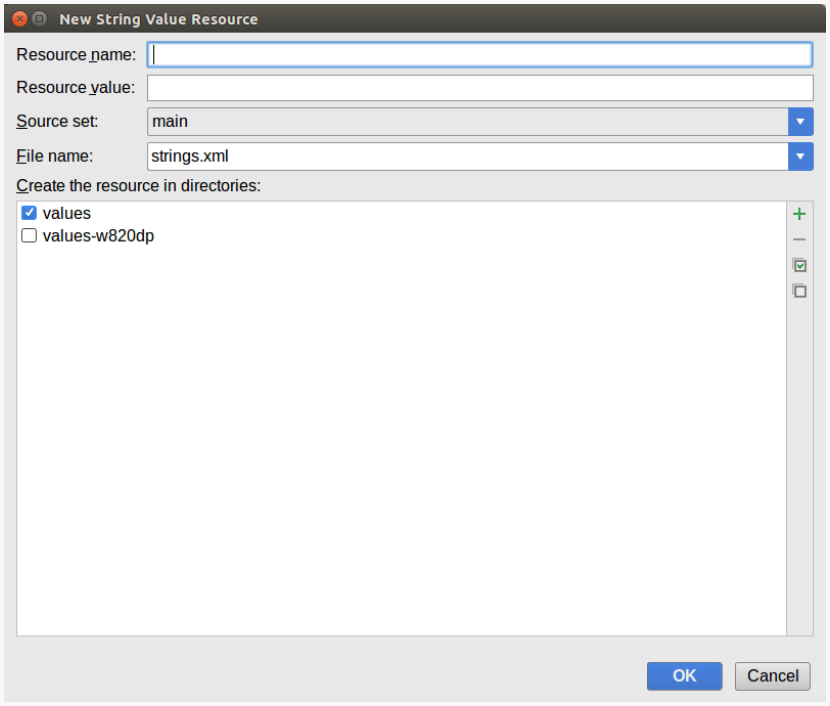


Figure 154: Android Studio New String Resource Dialog

Fill in a resource name of `hint` and a value of `Enter notes here`. Leave the rest of the dialog alone, and click OK.

Then, back in the attributes pane, switch to viewing all of the attributes, rather than just the subset. Scroll down to the `gravity` property, fold it open, and change the checked values to "top" and "start". In Android Studio 3.0, this will look bizarre, with four other checkmarks appearing, due to a bug in the IDE.

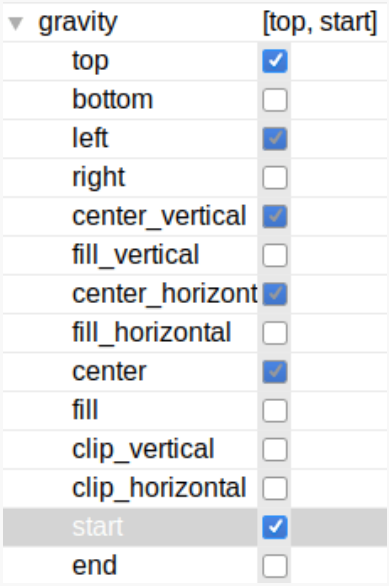


Figure 155: Layout Editor Attributes Pane, Showing Gravity Options

If you look at the layout XML in the Text sub-tab, you should have something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="match_parent"
  android:layout_height="match_parent">

  <EditText
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:inputType="textMultiLine"
    android:ems="10"
    android:id="@+id/editor"
    android:hint="@string/hint"
    android:gravity="top|start" />
</LinearLayout>
```

(from EmPubLite-AndroidStudio/T5-Layout/EmPubLite/app/src/main/res/layout/editor.xml)

Your hint may appear to be `"Enter notes here"`, as if you had directly typed that in rather than creating a string resource. As was covered earlier, Android Studio is lying to you. Click on the `"Enter notes here"` to see the actual string resource reference.

In Our Next Episode…

… we will attach a third-party library to our tutorial project.

GUI Building, Continued

If you are using an IDE, and you have been experimenting with the graphical layout editor and drag-and-drop GUI building, this chapter will cover some other general features of this editor that you may find useful.

Even if you are not using an IDE, you may want to at least skim this chapter, as you will find a few tricks that will be relevant for you as well.

Making Your Selection

Clicking on a widget makes it the selected widget, meaning that the toolbar buttons will affect that widget (or, sometimes, its container, depending upon the button). Selected widgets have a thin blue border with blue square "grab handles" for adjusting its size and position.

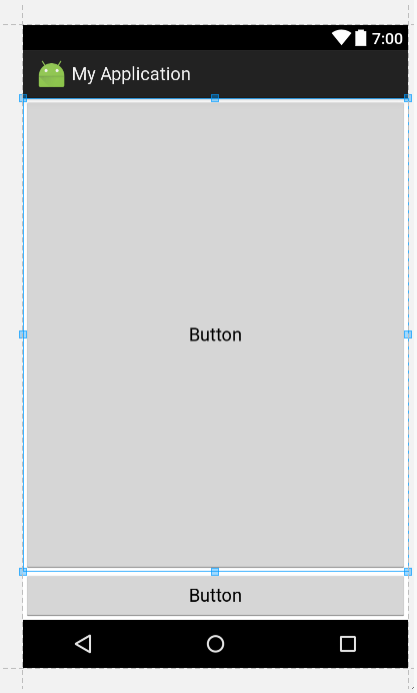


Figure 156: Android Studio, Selected Widget in Graphical Layout Editor

Clicking on a container also selects it. However, there may or may not be a blue border — in particular, containers that fill the screen (`match_parent` for width and height) do not seem to get the border.

Sometimes, though, you want to select a container that you cannot reach, because its contents are completely filled with widgets. In these cases, click on the widget or container in the Component Tree pane to select it.

Including Includes

Sometimes, you have a widget or a collection of widgets that you want to reuse across multiple layout XML resources. Android supports the notion of an "include" that allows this. Simply create a dedicated layout XML resource that contains the widget(s) to reuse, then add them to your main layouts via an `<include>` element:

```
<include layout="@layout/thing_we_are_reusing" />
```

You can even assign the `<include>` element a width or height if needed, as if it were just a widget or container.

The IDE used to make it easy for you to take widgets from an existing layout XML resource and extract them into a separate layout XML resource, replacing them with an `<include>` element. Multi-select the widgets in the blueprint or design views, or in the Preview tool when on the Text sub-tab of the layout editor. Then, right-click and choose

Refactor > Extract Layout from the context menu. This will display a dialog where you can fill in the file name of your resulting resource:

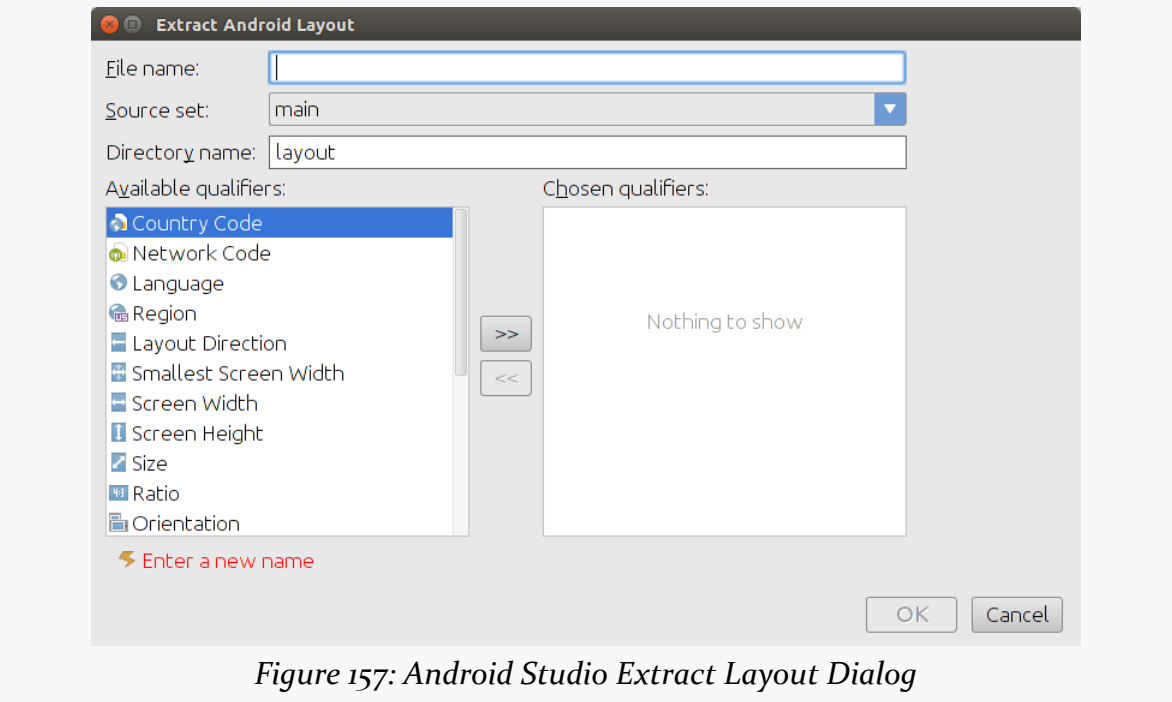


Figure 157: Android Studio Extract Layout Dialog

If you are extracting multiple widgets that are not wrapped in their own container, the IDE will automatically wrap them in a `<merge>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- widgets go here -->
</merge>
```

This is necessary purely from an XML standpoint — you cannot have multiple root elements in an XML file. When the `<merge>` is added to another layout via `<include>`, the `<merge>` element itself evaporates, leaving behind its children.

NOte that there is a bug that prevents this feature from working if you multi-select the widgets in the Component Tree.

Preview of Coming Attractions

At the top of the graphical layout editor, you will find a series of drop-downs that allow you to tailor what the preview looks like:

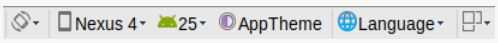


Figure 158: Android Studio Preview Controls

Your IDE will choose some likely defaults based upon your project settings, but you are welcome to change them as you see fit. Notable changes include:

- What version of Android is used for the preview (as widget styling changes from time to time in Android releases)
- What language is used for your string resources?
- What size and resolution of screen is used?
- Is it displayed in portrait or landscape?

These only affect the preview, so they show you (approximately) what your layout will look like under those conditions, but they do not modify anything about your layout XML itself.

AdapterViews and Adapters

If you want the user to choose something out of a collection of somethings, you could use a bunch of `RadioButton` widgets. However, Android has a series of more flexible widgets than that, ones that this book will refer to as "selection widgets".

These include:

- `ListView`, which is your typical "list box"
- `Spinner`, which (more or less) is a drop-down list
- `GridView`, offering a two-dimensional roster of choices
- `ExpandableListView`, a limited "tree" widget, supporting two levels in the hierarchy

and many more.

At their core, these are ordinary widgets. You will find them in your tool palette of your IDE's graphical layout editor, and can drag them and position them as you see fit.

The key is that these all have a common superclass: `AdapterView`, so named because they partner with objects implementing the `Adapter` interface to determine what choices are available for the user to choose from.

Adapting to the Circumstances

An `Adapter` is your bridge between your model data and that data's visual representation in the `AdapterView`:

- an `Adapter` might "adapt" an `Invoice` into a `View` that would serve as a row in a `ListView`
- an `Adapter` might "adapt" a `Book` into a `View` that would serve as a cell in a `GridView`
- and so on

Android ships with several `Adapter` classes ready for your use, where the different adapter classes are designed to "adapt" different sorts of collections (e.g., arrays versus results of database queries). Android also has a `BaseAdapter` class that can serve as the foundation for your own `Adapter` implementation, if you need to "adapt" a collection of data that does not fit any of the `Adapter` classes supplied by Android.

Using ArrayAdapter

The easiest adapter to use is `ArrayAdapter` — all you need to do is wrap one of these around a Java array or `java.util.List` instance, and you have a fully-functioning adapter:

```
String[] items={"this", "is", "a", "really", "silly", "list"};
new ArrayAdapter<String>(this,
                         android.R.layout.simple_list_item_1,
                         items);
```

One flavor of the `ArrayAdapter` constructor takes three parameters:

1. The `Context` to use (typically this will be your activity instance)
2. The resource ID of a view to use (such as a built-in system resource ID, as shown above)
3. The actual array or list of items to show

By default, the `ArrayAdapter` will invoke `toString()` on the objects in the list and wrap each of those strings in the view designated by the supplied resource. `android.R.layout.simple_list_item_1` simply turns those strings into `TextView` objects. Those `TextView` widgets, in turn, will be shown in the list or spinner or whatever widget uses this `ArrayAdapter`. If you want to see what `android.R.layout.simple_list_item_1` looks like, you can find a copy of it in your SDK installation — just search for `simple_list_item_1.xml`.

We will see in a later section how to subclass an `Adapter` and override row creation, to give you greater control over how rows and cells appear.

Lists of Naughty and Nice

The classic listbox widget in Android is known as `ListView`. Include one of these in your layout, invoke `setAdapter()` to supply your data and child views, and attach a listener via `setOnItemSelectedListener()` to find out when the selection has changed. With that, you have a fully-functioning listbox.

However, if your activity is dominated by a single list, you might well consider creating your activity as a subclass of `ListActivity`, rather than the regular `Activity` base class. If your main view is just the list, you do not even need to supply a layout — `ListActivity` will construct a full-screen list for you. If you do want to customize the layout, you can, so long as you identify your `ListView` as `@android:id/list`, so `ListActivity` knows which widget is the main list for the activity.

For example, here is a layout pulled from the `Selection/List` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    />
</LinearLayout>
```

(from Selection/List/app/src/main/res/layout/main.xml)

It is just a list with a label on top to show the current selection.

The Java code to configure the list and connect the list with the label is:

```
package com.commonsware.android.list;

import android.app.ListActivity;
import android.os.Bundle;
```

```
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;

public class ListViewDemo extends ListActivity {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                       android.R.layout.simple_list_item_1,
                       items));
    selection=(TextView)findViewById(R.id.selection);
  }

  @Override
  public void onListItemClick(ListView parent, View v, int position,
                              long id) {
    selection.setText(items[position]);
  }
}
```

(from Selection/List/app/src/main/java/com/commonsware/android/list/ListViewDemo.java)

With `ListActivity`, you can set the list adapter via `setListAdapter()` — in this case, providing an `ArrayAdapter` wrapping an array of Latin strings. To find out when the list selection changes, override `onListItemClick()` and take appropriate steps based on the supplied child view and position (in this case, updating the label with the text for that position).

The results?

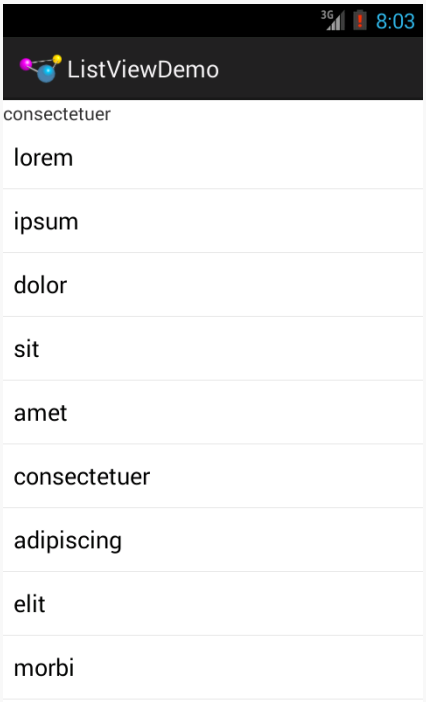


Figure 159: ListViewDemo, After User Taps on "consecteteur"

The second parameter to our `ArrayAdapter` — `android.R.layout.simple_list_item_1` — controls what the rows look like. The value used in the preceding example provides the standard Android list row: a big font with lots of padding to offer a large touch target for the user.

Clicks versus Selections

One thing that can confuse some Android developers is the distinction between clicks and selections. One might think that they are the same thing — after all, clicking on something selects it, right?

Well, no. At least, not in Android. At least not all of the time.

Android is designed to be used with touchscreen devices and non-touchscreen devices. Historically, Android has been dominated by devices that only offered touchscreens. However, there are various devices powered by Android and connected to TVs. Most TVs are not touchscreens, and so users of those TV-using Android devices will use some sort of remote control to drive Android. And some

Android devices offer both touchscreens and some other sort of pointing device — D-pad, trackball, arrow keys, etc.

To accommodate both styles of device, Android sometimes makes a distinction between selection events and click events. Widgets based on the "spinner" paradigm — including `Spinner` — treat everything as selection events. Other widgets — like `ListView` and `GridView` — treat selection events and click events differently. For these widgets, selection events are driven by the pointing device, such as using arrow keys to move a highlight bar up and down a list. Click events are when the user either "clicks" the pointing device (e.g., presses the center D-pad button) *or* taps on something in the widget using the touchscreen.

Choice Modes

By default, `ListView` is set up simply to collect clicks on list entries. Sometimes, though, you want a list that tracks a user's choice, or possibly multiple choices. `ListView` can handle that as well, but it requires a few changes.

First, you will need to call `setChoiceMode()` on the `ListView` in Java code to set the choice mode, classically supplying either `CHOICE_MODE_SINGLE` or `CHOICE_MODE_MULTIPLE` as the value. You can get your `ListView` from a `ListActivity` via `getListView()`. You can also declare this via the `android:choiceMode` attribute in your layout XML.

Then, rather than use `android.R.layout.simple_list_item_1` as the layout for the list rows in your `ArrayAdapter` constructor, you can use either `android.R.layout.simple_list_item_single_choice` or `android.R.layout.simple_list_item_multiple_choice` for single-choice or multiple-choice lists, respectively.

For example, here is an activity layout from the `Selection/Checklist` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:drawSelectorOnTop="false"
  android:choiceMode="multipleChoice"
/>
```

(from Selection/Checklist/app/src/main/res/layout/main.xml)

It is a full-screen `ListView`, with the `android:choiceMode="multipleChoice"` attribute to indicate that we want multiple choice support.

Our activity just uses a standard `ArrayAdapter` on our list of Latin words, but uses `android.R.layout.simple_list_item_multiple_choice` as the row layout:

```
package com.commonsware.android.checklist;

import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;

public class ChecklistDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_multiple_choice,
                        items));
  }
}
```

(from Selection/Checklist/app/src/main/java/com/commonsware/android/checklist/ChecklistDemo.java)

What the user sees is the list of words with checkboxes down the right edge:

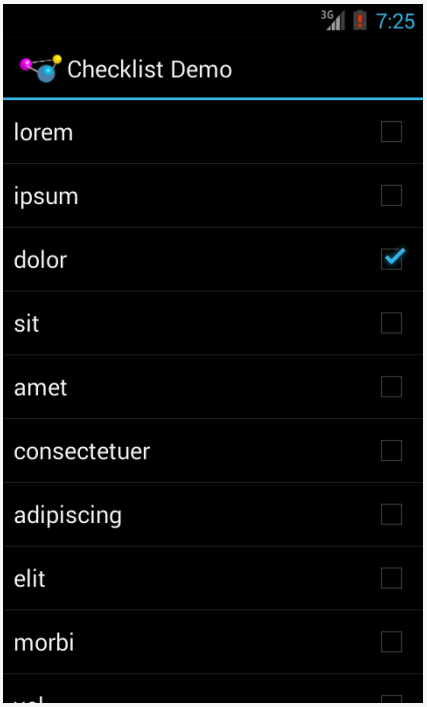


Figure 160: Multiple-Choice Mode

If we wanted, we could call methods like `getCheckedItemPositions()` on our `ListView` to find out which items the user checked, or `setItemChecked()` if we wanted to check (or un-check) a specific entry ourselves.

Clicks versus Selections, Revisited

If the user clicks a row in a `ListView`, a click event is registered, triggering things like `onListItemClick()` in an `OnItemClickListener`. If the user uses a pointing device to change a selection (e.g., pressing up and down arrows to move a highlight bar in the `ListView`), that triggers `onItemSelected()` in an `OnItemSelectedListener`.

Many times, particularly if the `ListView` is the entire UI at present, you only care about clicks. Sometimes, particularly if the `ListView` is adjacent to something else (e.g., on a TV, where you have more screen space *and* do not have a touchscreen), you will care more about selection events. Either way, you can get the events you need.

Spin Control

In Android, the `Spinner` is the equivalent of the drop-down selector you might find in other toolkits. Clicking the `Spinner` drops down a list for the user to choose an item from. You basically get the ability to choose an item from a list without taking up all the screen space of a `ListView`, at the cost of an extra click to make a change.

As with `ListView`, you provide the adapter for data and child views via `setAdapter()` and hook in a listener object for selections via `setOnItemSelectedListener()`.

To tailor the view used when displaying the drop-down perspective, you need to configure the adapter, not the `Spinner` widget. Use the `setDropDownViewResource()` method to supply the resource ID of the view to use.

For example, culled from the `Selection/Spinner` sample project, here is an XML layout for a simple view with a `Spinner`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

(from Selection/Spinner/app/src/main/res/layout/main.xml)

This is the same view as shown in a previous section, just with a `Spinner` instead of a `ListView`.

To populate and use the `Spinner`, we need some Java code:

```
public class SpinnerDemo extends Activity
  implements AdapterView.OnItemSelectedListener {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              items);

    aa.setDropDownViewResource(
      android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }

  @Override
  public void onItemSelected(AdapterView<?> parent,
                             View v, int position, long id) {
    selection.setText(items[position]);
  }

  @Override
  public void onNothingSelected(AdapterView<?> parent) {
    selection.setText("");
  }
}
```

(from Selection/Spinner/app/src/main/java/com/commonsware/android/selection/SpinnerDemo.java)

Here, we attach the activity itself as the selection listener (`spin.setOnItemSelectedListener(this)`), as `Spinner` widgets only support selection events, not click events. This works because the activity implements the `OnItemSelectedListener` interface. We configure the adapter not only with the list of fake words, but also with a specific resource to use for the drop-down view (via

`aa.setDropDownViewResource()`). Also note the use of `android.R.layout.simple_spinner_item` as the built-in `View` for showing items in the spinner itself. Finally, we implement the callbacks required by `OnItemSelectedListener` to adjust the selection label based on user input.

What we get is:

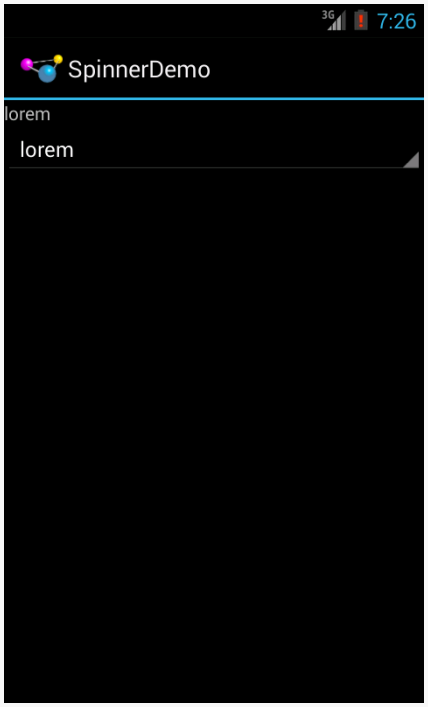


Figure 161: SpinnerDemo, as Initially Launched

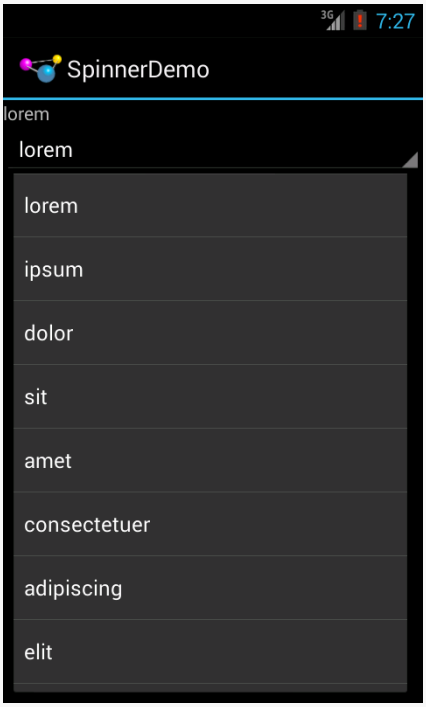


Figure 162: SpinnerDemo, with Spinner Drop-Down List Displayed

Grid Your Lions (Or Something Like That…)

As the name suggests, `GridView` gives you a two-dimensional grid of items to choose from. You have moderate control over the number and size of the columns; the number of rows is dynamically determined based on the number of items the supplied adapter says are available for viewing.

There are a few properties which, when combined, determine the number of columns and their sizes:

1. `android:numColumns` spells out how many columns there are, or, if you supply a value of `auto_fit`, Android will compute the number of columns based on available space and the properties listed below.
2. `android:verticalSpacing` and `android:horizontalSpacing` indicate how much whitespace there should be between items in the grid.
3. `android:columnWidth` indicates how wide each column should be, in terms of some dimension value (e.g., `40dp` or `@dimen/grid_column_width`).
4. `android:stretchMode` indicates, for grids with `auto_fit` for `android:numColumns`, what should happen for any available space not taken

up by columns or spacing — this should be `columnWidth` to have the columns take up available space or `spacingWidth` to have the whitespace between columns absorb extra space.

Otherwise, the `GridView` works much like any other selection widget — use `setAdapter()` to provide the data and child views, invoke `setOnItemClickListener()` to find out when somebody clicks on a cell in the grid, etc.

For example, here is an XML layout from the `Selection/Grid` sample project, showing a `GridView` configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <GridView
    android:id="@+id/grid"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:verticalSpacing="40dip"
    android:horizontalSpacing="5dip"
    android:numColumns="auto_fit"
    android:columnWidth="100dip"
    android:stretchMode="columnWidth"
    android:gravity="center"
    />
</LinearLayout>
```

(from Selection/Grid/app/src/main/res/layout/main.xml)

For this grid, we take up the entire screen except for what our selection label requires. The number of columns is computed by Android (`android:numColumns = "auto_fit"`) based on our horizontal spacing (`android:horizontalSpacing = "5dip"`) and columns width (`android:columnWidth = "100dip"`), with the columns absorbing any "slop" width left over (`android:stretchMode = "columnWidth"`).

The Java code to configure the `GridView` is:

```
package com.commonsware.android.grid;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.GridView;
import android.widget.TextView;

public class GridDemo extends Activity
  implements AdapterView.OnItemClickListener {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    GridView g=(GridView) findViewById(R.id.grid);
    g.setAdapter(new ArrayAdapter<String>(this,
                        R.layout.cell,
                        items));
    g.setOnItemClickListener(this);
  }

  @Override
  public void onItemClick(AdapterView<?> parent, View v,
                              int position, long id) {
    selection.setText(items[position]);
  }
}
```

(from Selection/Grid/app/src/main/java/com/commonsware/android/grid/GridDemo.java)

The grid cells are defined by a separate `res/layout/cell.xml` file, referenced in our `ArrayAdapter` as `R.layout.cell`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:textSize="14dip"
/>
```

(from Selection/Grid/app/src/main/res/layout/cell.xml)

With the vertical spacing from the XML layout (`android:verticalSpacing = "40dip"`), the grid overflows the boundaries of the emulator's screen:

Figure 163: GridDemo, as Initially Launched

Figure 164: GridDemo, Scrolled to the Bottom of the Grid

`GridView`, like `ListView`, supports both click events and selection events. In this sample, we register an `OnItemClickListener` to listen for click events.

Fields: Now With 35% Less Typing!

The `AutoCompleteTextView` is sort of a hybrid between the `EditText` (field) and the `Spinner`. With auto-completion, as the user types, the text is treated as a prefix filter, comparing the entered text as a prefix against a list of candidates. Matches are shown in a selection list that folds down from the field. The user can either type out an entry (e.g., something not in the list) or choose an entry from the list to be the value of the field.

`AutoCompleteTextView` subclasses `EditText`, so you can configure all the standard look-and-feel aspects, such as font face and color.

In addition, `AutoCompleteTextView` has an `android:completionThreshold` property, to indicate the minimum number of characters a user must enter before the list filtering begins.

You can give `AutoCompleteTextView` an adapter containing the list of candidate values via `setAdapter()`. However, since the user could type something not in the list, `AutoCompleteTextView` does not support selection listeners. Instead, you can register a `TextWatcher`, like you can with any `EditText`, to be notified when the text changes. These events will occur either because of manual typing or from a selection from the drop-down list.

Below we have a familiar-looking XML layout, this time containing an `AutoCompleteTextView` (pulled from the `Selection/AutoComplete` sample application):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <AutoCompleteTextView android:id="@+id/edit"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:completionThreshold="3"/>
</LinearLayout>
```

(from Selection/AutoComplete/app/src/main/res/layout/main.xml)

The corresponding Java code is:

```
package com.commonsware.android.auto;

import android.app.Activity;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;
import android.widget.TextView;

public class AutoCompleteDemo extends Activity
  implements TextWatcher {
  private TextView selection;
```

```
  private AutoCompleteTextView edit;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);
    edit=(AutoCompleteTextView)findViewById(R.id.edit);
    edit.addTextChangedListener(this);

    edit.setAdapter(new ArrayAdapter<String>(this,
                          android.R.layout.simple_dropdown_item_1line,
                          items));
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                            int count) {
    selection.setText(edit.getText());
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start,
                                int count, int after) {
    // needed for interface, but not used
  }

  @Override
  public void afterTextChanged(Editable s) {
    // needed for interface, but not used
  }
}
```

(from Selection/AutoComplete/app/src/main/java/com/commonsware/android/auto/AutoCompleteDemo.java)

This time, our activity implements `TextWatcher`, which means our callbacks are `onTextChanged()`, `beforeTextChanged()`, and `afterTextChanged()`. In this case, we are only interested in the first, and we update the selection label to match the `AutoCompleteTextView`'s current contents.

Here we have the results:

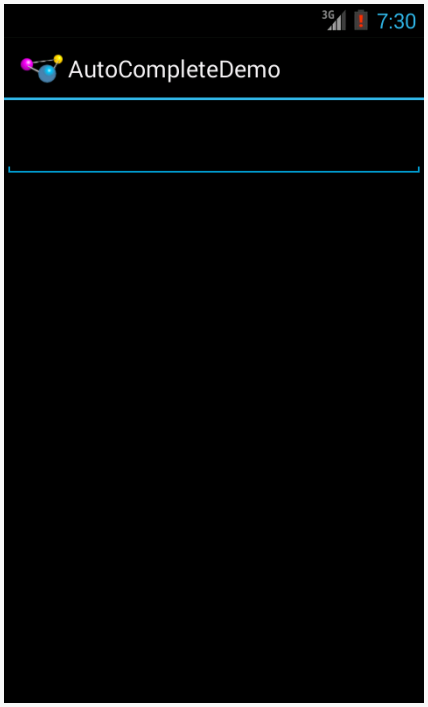


Figure 165: AutoCompleteDemo, as Initially Launched

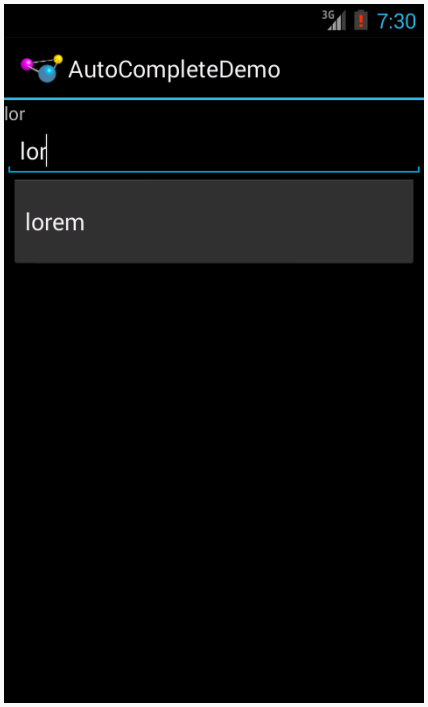


Figure 166: AutoCompleteDemo, After Entering a Few Matching Letters

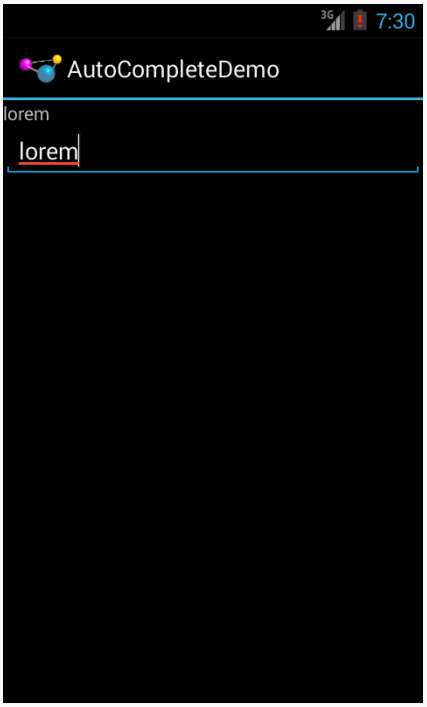


Figure 167: AutoCompleteDemo, After Auto-Complete Value Was Selected

Note that the red underline in the preceding screenshot is due to spelling correction. Like `EditText`, `AutoCompleteTextView` supports hinting at spelling errors. The emulator's language is set to English, as there is no option in it for Latin.

Customizing the Adapter

The humble `ListView` is one of the most important widgets in all of Android, simply because it is used so frequently. Whether choosing a contact to call or an email message to forward or an ebook to read, `ListView` widgets are employed in a wide range of activities.

Of course, it would be nice if they were more than just plain text.

The good news is that they can be as fancy as you want, within the limitations of a mobile device's screen, of course. However, making them more elaborate takes some work.

Note that while this section will be using `ListView` as the `AdapterView`, the same techniques hold for *any* `AdapterView`.

The Single Layout Pattern

The simplest way of creating custom `ListView` rows (or `GridView` cells or whatever) is when they all have the same basic structure and can be created from the same layout XML resource. This does not mean they have to be strictly identical, but that you can make whatever changes you need just by configuring the widgets (e.g., make some things `VISIBLE` or `GONE`).

This is not especially difficult, though it does take a few more steps than what we have seen previously.

Step #0: Get Things Set Up Simply

First, create your activity (e.g., `ListActivity`), get your data (e.g., array of Java strings), and set up your `AdapterView` with a simple adapter following the steps outlined in the preceding sections.

Here, we will examine the `Selection/Dynamic` sample project. We will use a simple `ListActivity` (taking the default layout of a full-screen `ListView`) and use the same list of 25 Latin words used in earlier samples. However, this time, we want to have a more elaborate row, taking into account the length of the Latin word.

Step #1: Design Your Row

Next, create a layout XML resource that will represent one row in your `ListView` (or cell in your `GridView` or whatever).

For example, our `res/layout/row.xml` resource will use a pair of nested `LinearLayout` containers to organize two `TextView` widgets and an `ImageView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:padding="2dip"
```

```
    android:src="@drawable/ok"
    android:contentDescription="@string/icon"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/size"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

(from Selection/Dynamic/app/src/main/res/layout/row.xml)

The `ImageView` will use one of two drawable resources, one for short words, and another for long words.

Step #2: Extend ArrayAdapter

If you just used `R.layout.row` with a regular `ArrayAdapter`, it would work, insofar as it would not crash. However, `ArrayAdapter` only knows how to update a single `TextView` in a row, so it would ignore our other `TextView`, let alone the `ImageView`.

So, we need to create our own `ListAdapter`, by creating our own subclass of `ArrayAdapter`.

Since an `Adapter` is tightly coupled to the `AdapterView` that uses it, it is typically simplest to make the custom `ArrayAdapter` subclass be an inner class of whoever manages the `AdapterView`. Hence, in our sample, we will create an `IconicAdapter` inner class of our `ListActivity`.

Step #3: Override the Constructor and `getView()`

The `IconicAdapter` constructor can chain to the superclass and supply the necessary data, such as our Java array of Latin words. The real fun comes when we override `getView()`:

```
package com.commonsware.android.fancylists.three;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;

public class DynamicDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setListAdapter(new IconicAdapter());
  }

  class IconicAdapter extends ArrayAdapter<String> {
    IconicAdapter() {
      super(DynamicDemo.this, R.layout.row, R.id.label, items);
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      if (items[position].length()>4) {
        icon.setImageResource(R.drawable.delete);
      }
      else {
        icon.setImageResource(R.drawable.ok);
      }

      TextView size=(TextView)row.findViewById(R.id.size);

      size.setText(String.format(getString(R.string.size_template), items[position].length()));

      return(row);
    }
  }
}
```

(from Selection/Dynamic/app/src/main/java/com/commonsware/android/fancylists/three/DynamicDemo.java)

Our `getView()` implementation does three things:

- It chains to the superclass' implementation of `getView()`, which returns to us an instance of our row `View`, as prepared by `ArrayAdapter`. In particular, our word has already been put into one `TextView`, since `ArrayAdapter` does that normally.
- It finds our `ImageView` and applies a business rule to set which icon should be used, referencing one of two drawable resources (`R.drawable.ok` and `R.drawable.delete`).
- It finds our other `TextView` and populates it as well, by pulling in the value of a string resource and using `String.format()` to pour in our word length.

Note that we call `findViewById()` not on the activity, but rather on the row returned by the superclass' implementation of `getView()`. **Always call `findViewById()` on something that is guaranteed to give you a unique result.** In the case of an `AdapterView`, there will be many rows, cells, etc. — calling `findViewById()` on the activity might return widgets with the right name but from other rows or cells.

This gives us:

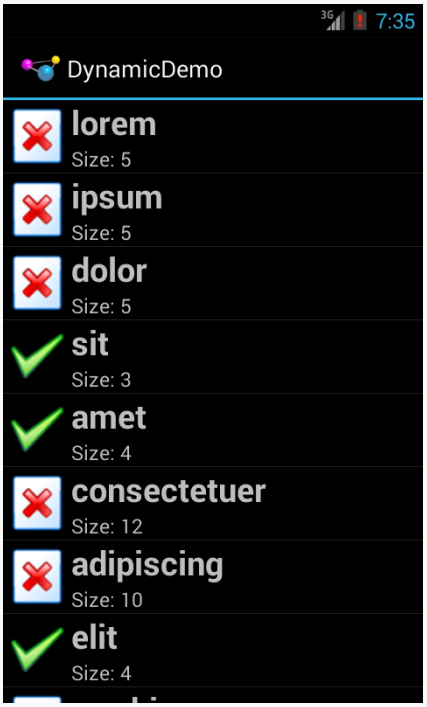


Figure 168: The Dynamic Sample Application

The approach of overriding `getView()` works for `ArrayAdapter`, but some other types of adapters would have alternatives. We will see that mostly with `CursorAdapter`, profiled in upcoming chapters.

Optimizing with the ViewHolder Pattern

A somewhat expensive operation we do a lot with more elaborate list rows is call `findViewById()`. This dives into our row and pulls out widgets by their assigned identifiers, so we can customize the widget contents (e.g., change the text of a `TextView`, change the icon in an `ImageView`). Since `findViewById()` can find widgets anywhere in the tree of children of the row's root `View`, this could take a fair number of instructions to execute, particularly if we keep having to re-find widgets we had found once before.

In some GUI toolkits, this problem is avoided by having the composite `View` objects, like our rows, be declared totally in program code (in this case, Java). Then, accessing individual widgets is merely the matter of calling a getter or accessing a field. And you can certainly do that with Android, but the code gets rather verbose. What would be nice is a way where we can still use the layout XML yet cache our row's key child widgets so we only have to find them once.

That's where the holder pattern comes into play, in a class we will call `ViewHolder`.

All `View` objects have `getTag()` and `setTag()` methods. These allow you to associate an arbitrary object with the widget. What the holder pattern does is use that "tag" to hold an object that, in turn, holds each of the child widgets of interest. By attaching that holder to the row `View`, every time we use the row, we already have access to the child widgets we care about, without having to call `findViewById()` again.

So, let's take a look at one of these holder classes (taken from the `Selection/ViewHolder` sample project, a revised version of the `Selection/Dynamic` sample from before):

```
package com.commonsware.android.fancylists.five;

import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;

class ViewHolder {
  ImageView icon=null;
  TextView size=null;

  ViewHolder(View row) {
    this.icon=(ImageView)row.findViewById(R.id.icon);
    this.size=(TextView)row.findViewById(R.id.size);
  }
}
```

(from Selection/ViewHolder/app/src/main/java/com/commonsware/android/fancylists/five/ViewHolder.java)

`ViewHolder` holds onto the child widgets, initialized via `findViewById()` in its constructor. The widgets are simply package-protected data members, accessible from other classes in this project... such as a `ViewHolderDemo` activity. In this case, we are only holding onto two widgets — the icon and the second label – since we will let `ArrayAdapter` handle our first label for us. In our case, we are holding onto the `TextView` and `ImageView` widgets that we want to populate in `getView()`.

Using `ViewHolder` is a matter of creating an instance whenever we inflate a row and attaching said instance to the row `View` via `setTag()`, as shown in this rewrite of `getView()`, found in `ViewHolderDemo`:

```
    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
```

```
    ViewHolder holder=(ViewHolder)row.getTag();

    if (holder==null) {
      holder=new ViewHolder(row);
      row.setTag(holder);
    }

    if (getModel(position).length()>4) {
      holder.icon.setImageResource(R.drawable.delete);
    }
    else {
      holder.icon.setImageResource(R.drawable.ok);
    }

    holder.size.setText(String.format(getString(R.string.size_template), items[position].length()));

    return(row);
  }
```

(from Selection/ViewHolder/app/src/main/java/com/commonsware/android/fancylists/five/ViewHolderDemo.java)

If the call to `getTag()` on the row returns `null`, we know we need to create a new `ViewHolder`, which we then attach to the row via `setTag()` for later reuse. Then, accessing the child widgets is merely a matter of accessing the data members on the holder.

This takes advantage of the fact that rows in a `ListView` get *recycled* – a 25,000-row list does not create 25,000 rows. The recycling itself is handled for us by `ArrayAdapter`, so we simply have to create our `ViewHolder` when needed and reuse the existing `ViewHolder` when a row gets recycled. The first time the `ListView` is displayed, all new rows need to be created, and we wind up creating a `ViewHolder` for each. As the user scrolls, rows get recycled, and we can reuse their corresponding `ViewHolder` widget caches. We will cover this recycling process in greater detail in a later chapter.

Note that the `getModel()` method shown here retrieves our model `String` for a given `position`, by using `getListAdapter()` (to retrieve our `IconicAdapter` from the activity's `ListView`) and `getItem()` (to retrieve the data, held by the adapter, represented by the position):

```
  private String getModel(int position) {
    return(((IconicAdapter)getListAdapter()).getItem(position));
  }
```

(from Selection/ViewHolder/app/src/main/java/com/commonsware/android/fancylists/five/ViewHolderDemo.java)

Dealing with Multiple Row Layouts

The story gets significantly more complicated if our mix of rows is more complicated. For example, here is the Sound screen in the Settings application:

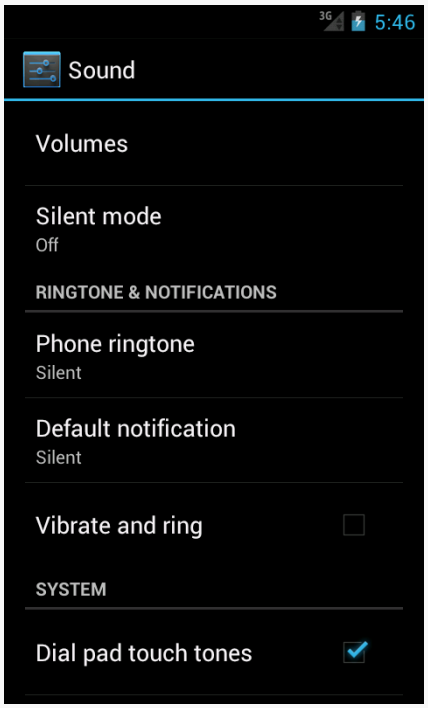


Figure 169: Sound Settings Screen

It may not look like it, but that is a `ListView`. However, not all the rows look the same:

- Some have one line of text (e.g., "Volumes")
- Some have two lines of text (e.g., "Silent mode" plus "Off")
- Some have one line of text and a `CheckBox` (e.g., "Vibrate and ring")
- Some are headings with totally different text formatting (e.g., "RINGTONE & NOTIFICATIONS")

This is handled by having more than one row layout XML resource used by the adapter. The complexity comes not only in managing those different resources and determining which to use when, but in just having more than one resource – after all, we only teach `ArrayAdapter` how to use one. We will examine how to handle this scenario in a later chapter.

Visit the Trails!

To learn more about `ListView`, you can turn to Advanced ListViews, which covers other tricks you can do with a `ListView`.

`RecyclerView` is a more powerful (and more complex) replacement for `ListView` and `GridView`. You can read more about what it does and how you can use it.

The WebView Widget

HTML has come a *long* way from Sir Tim Berners-Lee's original vision of using it to publish physics papers.

Not surprisingly, displaying HTML, CSS, and JavaScript in mobile applications is fairly popular, not only for creating full-fledged Web browsers, but for rendering HTML content from RSS/Atom feeds, from HTML-formatted email messages, ebooks (like the one you are reading), and so forth.

There are a couple of ways to display HTML in Android, with the most powerful being the `WebView` widget, the focus of this chapter.

Role of WebView

If your HTML is fairly limited in scope, such as what you might find in the body of a status update on Twitter, you can use the static `fromHtml()` method on the `Html` utility class to parse an HTML-formatted string into something that you can put into a `TextView`. `TextView` can render simple formatting like styles (bold, italic, etc.), font faces (serif, sans serif, etc.), colors, links, and so forth.

However, sometimes your needs for HTML transcend what `TextView` can handle. You will not be browsing Facebook using `TextView`, for example.

In those cases, `WebView` will be the more appropriate widget, as it can handle a *much* wider range of HTML tags. `WebView` can also handle CSS and JavaScript, which `Html.fromHtml()` would simply ignore. `WebView` can also assist you with common "browsing" metaphors, such as history list of visited URLs to support backwards and forwards navigation.

On the other hand, `WebView` is a much more expensive widget to use, in terms of memory consumption, than is `TextView`.

Daddy, Where Do WebViews Come From?

Originally, the story was simple: `WebView` was powered by a fairly complete copy of WebKit, the Web rendering engine behind Safari and, originally, Chrome.

In Android 4.4, Google switched rendering engines. Depending on who you asked, `WebView` was powered by Chromium or Blink. Chromium is an open source browser that forms the foundation for Google's Chrome, and Blink is a fork of WebKit created by Opera and Google that, in turn, powers Chromium.

Starting in Android 5.0, the implementation of `WebView` was no longer a part of Android. Rather, it became a separate "System WebView" app, distributed through the Play Store. The idea was that this app could be updated independently of the device firmware, so that `WebView` bugs could be fixed more rapidly and distributed to more devices. This also means that Google can distribute new and exciting bugs more quickly (and independently of Android OS version), as will be discussed later in the chapter.

In Android 7.0, the implementation of `WebView` will be from one of two places:

- the proprietary Chrome browser app, or
- the System WebView app, for devices where Chrome is disabled

The documented dependency of `WebView` on apps distributed through the Play Store makes things very murky for non-Play ecosystem devices, such as most devices in China. Most likely, individual manufacturers do their own thing with respect to updating `WebView`.

As a result, from the standpoint of security and compatibility, `WebView` is a "hot mess".

Adding the Widget

For simple stuff, `WebView` is not significantly different than any other widget in Android — pop it into a layout, tell it what URL to navigate to via Java code, and you are done.

As you can see in the WebKit/Browser1 sample application, here is a simple layout with a WebView:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/webkit"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

(from WebKit/Browser1/app/src/main/res/layout/main.xml)

As with any other widget, you need to tell it how it should fill up the space in the layout (in this case, it fills all remaining space).

And, just as with other widgets, you can add it using your IDE's graphical layout editor. An Android Studio user can drag a WebView out of the "Containers" section of the tool palette.

Note that WebView knows how to scroll its own contents, so you do not need to put it in a ScrollView or HorizontalScrollView.

Loading Content Via a URL

There are a number of ways to load HTML content into a WebView widget.

The simplest is to use the loadUrl() method, which takes a URL and retrieves its contents over the Internet. For example, here is the activity source code for the WebKit/Browser1 sample application:

```
package com.commonsware.android.browser1;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class BrowserDemo1 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);
```

```
    browser.loadUrl("https://commonsware.com");
  }
}
```

(from WebKit/Browser1/app/src/main/java/com/commonsware/android/browser1/BrowserDemo1.java)

However, we also have to make one change to `AndroidManifest.xml`, adding a line where we request permission to access the Internet:

```
  <uses-permission android:name="android.permission.INTERNET"/>
```

(from WebKit/Browser1/app/src/main/AndroidManifest.xml)

If we fail to add this permission, the browser will refuse to load pages. We will discuss more about this "permission" concept in a later chapter.

The resulting activity looks like a Web browser, just with hidden scrollbars:

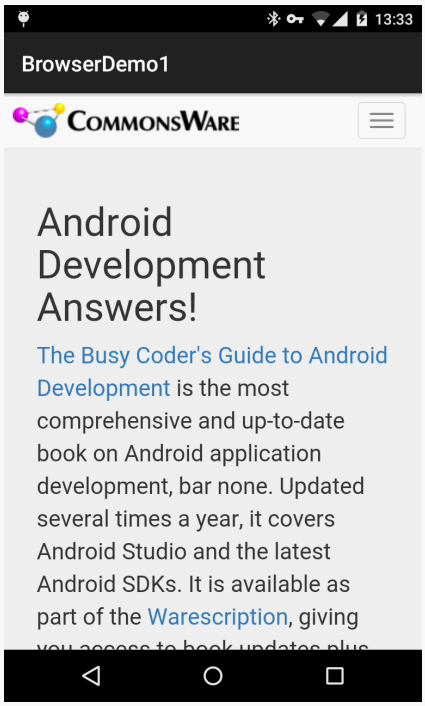


Figure 170: The Browser1 Sample Application (image from June 2015)

As with a regular Android Web browser, you can pan around the page by dragging it, while the directional pad moves you around all the focusable elements on the page.

What is missing is all the extra stuff that make up a Web browser, such as a navigational toolbar. `WebView` does not provide any of that — if you want those sorts of UI features, you will need to implement those yourself (e.g., use an `EditText` or `AutoCompleteTextView` for a browser address bar).

Links and Redirects

The sample shown above loads the CommonsWare home page. The links in that page are clickable. Exactly what happens when you click on the link, though, depends upon circumstances.

Traditionally, the default behavior for when the user clicks on a link in a `WebView` is for the linked-to Web page to be launched in a Web browser. However, the "Android System WebView" released in early June 2015 changed that default behavior, so now the linked-to Web page opens up in the `WebView` itself. Since Android 4.4 and older devices do not have the "Android System WebView", this means that the default behavior of link clicks varies by device, which is not fun.

Also, if you try loading a page using `loadUrl()`, and the server issues a server-side redirect (e.g., HTTP 301 or 304 response), the default behavior is the same as a simple click of a link:

- On devices with "Android System WebView" 43.0.2357.121 or newer, the redirected-to page shows up in the `WebView`
- Everywhere else, the redirected-to page appears in a separate Web browser app

We will cover how to address this problem later in this chapter.

Supporting JavaScript

Now, you may be tempted to replace the URL in the above source code with something else, such as Google's home page or something else that relies upon JavaScript. You will find that such pages do not work especially well by default. That is because, by default, JavaScript is turned off in `WebView` widgets.

If you want to enable JavaScript, call `getSettings().setJavaScriptEnabled(true);` on the `WebView` instance. At this point, any JavaScript referenced by your Web page should work normally.

There are some fancy tricks you can perform with `WebView` and JavaScript, such as having JavaScript call Java code or vice versa. These techniques will be covered in a later chapter.

Alternatives for Loading Content

`loadUrl()` works with:

- `http://` and `https://` URLs
- `file://` URLs pointing to the local filesystem
- `file:///android_asset/` URLs pointing to one of your application's assets, as will be discussed later in this book
- `content://` URLs pointing to a `ContentProvider` that is publishing content available for streaming, as will be discussed much later in this book

Instead of `loadUrl()`, you can also use `loadData()`. Here, you supply the HTML for the `WebView` to display. You might use this to:

1. display a manual that was installed as a file with your application package
2. display snippets of HTML you retrieved as part of other processing, such as the description of an entry in an Atom feed
3. generate a whole user interface using HTML, instead of using the Android widget set

There are two flavors of `loadData()`. The simpler one allows you to provide the content, the MIME type, and the encoding, all as strings. Typically, your MIME type will be `text/html` and your encoding will be `UTF-8` for ordinary HTML.

For example, if you replace the `loadUrl()` invocation in the previous example with the following:

```
browser.loadData("<html><body>Hello, world!</body></html>",
                 "text/html", "UTF-8");
```

You get:

Figure 171: The Browser2 sample application

This is also available as a fully-buildable sample, as `WebKit/Browser2`.

There is also a `loadDataWithBaseURL()` method. This takes, among other parameters, the "base URL" to use when resolving relative URLs in the HTML. Any relative URL (e.g., `<img src="images/foo.png">`) will be interpreted as being relative to the base URL supplied to `loadDataWithBaseURL()`. If you find that you have content that refuses to load properly with `loadData()`, try `loadDataWithBaseURL()` with a `null` base URL, as sometimes that works better, for unknown reasons.

Listening for Events

Particularly if you are going to use the `WebView` as a local user interface (vs. browsing the Web), you will want to be able to get control at key times, particularly when users click on links. You will want to make sure those links are handled properly, either by loading your own content back into the `WebView`, by submitting an `Intent` to Android to open the URL in a full browser, or by some other means. We will discuss using an `Intent` to launch a Web browser in a later chapter.

One hook into the `WebView` activity is via `setWebViewClient()`, which takes an instance of a `WebViewClient` implementation as a parameter. The supplied callback object will be notified of a wide range of events, ranging from when parts of a page have been retrieved (`onPageStarted()`, etc.) to when you, as the host application, need to handle certain user- or circumstance-initiated events, such as:

1. `onTooManyRedirects()`
2. `onReceivedHttpAuthRequest()`
3. etc.

A common hook will be `shouldOverrideUrlLoading()`, where your callback is passed a URL (plus the `WebView` itself) and you return `true` if you will handle the request or `false` if you want default handling (e.g., actually fetch the Web page referenced by the URL). In the case of a feed reader application, for example, you will probably not have a full browser with navigation built into your reader, so if the user clicks a URL, you probably want to use an `Intent` to ask Android to load that page in a full browser. But, if you have inserted a "fake" URL into the HTML, representing a link to some activity-provided content, you can update the `WebView` yourself.

For example, let's amend the first browser example to be an application that, upon a click, shows the current time.

From [WebKit/Browser3], here is the revised Java:

```
package com.commonsware.android.webkit;

import android.app.Activity;
import android.os.Bundle;
import android.text.format.DateUtils;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import java.util.Date;

public class BrowserDemo3 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);
    browser.setWebViewClient(new Callback());
```

```
    loadTime();
  }

  void loadTime() {
    String page=
        "<html><body><a href='http://webview.used.to.be.less.annoying/clock'>"
            + DateUtils.formatDateTime(this, new Date().getTime(),
                                       DateUtils.FORMAT_SHOW_DATE
                                           | DateUtils.FORMAT_SHOW_TIME)
            + "</a></body></html>";

    browser.loadData(page, "text/html", "UTF-8");
  }

  private class Callback extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
      loadTime();

      return(true);
    }
  }
}
```

Here, we load a simple Web page into the browser (`loadTime()`) that consists of the current time, made into a hyperlink to a fake URL. We also attach an instance of a `WebViewClient` subclass, providing our implementation of `shouldOverrideUrlLoading()`. In this case, no matter what the URL, we want to just reload the `WebView` via `loadTime()`.

Running this activity gives us:

Figure 172: The Browser3 Sample Application

Clicking the link will cause us to rebuild the page with the new time.

Note that we are using a `DateUtils` utility class supplied by Android for formatting our date and time. The big advantage of using `DateUtils` is that this class is aware of the user's settings for how they prefer to see the date and time (e.g., 12- versus 24-hour mode).

There is also a `WebChromeClient` that you can register with a `WebView` via a call to `setWebChromeClient()`. This object will be called when various things occur in the `WebView` that might pertain to a browser's "chrome" (i.e., the things outside the HTML rendering area). For example, `onJSAlert()` will be called on your `WebChromeClient` when JavaScript code calls `alert()`.

Addressing the Link/Redirect Behavior

Given that Google, through "Android System WebView" 43.0.2357.121, has changed the default behavior for when users click on links or redirects, it is in your best interests to *avoid the default*, since the default varies.

To do this, you can use `WebViewClient` and `shouldOverrideUrlLoading()`, as indicated above.

The WebKit/Browser4 is a clone of the original sample from this chapter, with one change: adding in a `WebViewClient` to force all link clicks to alter the `WebView` contents, regardless of what version of Android or the "Android System WebView" we are using:

```
package com.commonsware.android.browser4;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;
import android.webkit.WebViewClient;

public class BrowserDemo4 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    browser.setWebViewClient(new WebViewClient() {
      @Override
      public boolean shouldOverrideUrlLoading(WebView view, String url) {
        view.loadUrl(url);

        return(true);
      }
    });

    browser.loadUrl("http://commonsware.com");
  }
}
```

(from WebKit/Browser4/app/src/main/java/com/commonsware/android/browser4/BrowserDemo4.java)

Here, the `WebViewClient` is an instance of an anonymous inner class, and `shouldOverrideUrlLoading()` just turns around and calls `loadUrl()` on the `WebView` to handle the new URL. `shouldOverrideUrlLoading()` returns `true` to indicate that it is handling the event.

Visit the Trails!

You can learn more about powerful tricks with `WebView`, including integrating the Java and JavaScript environments, in a later chapter.

Defining and Using Styles

As noted in an earlier chapter, Android offers styles and themes, filling the same sort of role that CSS does in Web development. In that earlier chapter, we covered the basic roles of styles and themes, plus introduced the three classic theme families:

- `Theme`
- `Theme.Holo`
- `Theme.Material`

In this chapter, we will take a slightly "deeper dive" into styles and themes, exploring how you can create your own and apply them to your app's UI.

Styles: DIY DRY

The purpose of styles is to encapsulate a set of attributes that you intend to use repeatedly, conditionally, or otherwise wish to keep separate from your layouts proper. The primary use case is "don't repeat yourself" (DRY) — if you have a bunch of widgets that look the same, use a style to use a single definition for "look the same", rather than copying the look from widget to widget.

And that paragraph will make a bit more sense if we look at an example, specifically the `Styles/NowStyled` sample project. This is a trivial project, with a full-screen button that shows the date and time of when the activity was launched or when the button was pushed. This time, though, we want to change the way the text on the face of the button appears, and we will do so using a style.

The `res/layout/main.xml` file in this project has a `style` attribute on the `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/button"
  android:text=""
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  style="@style/bigred"
/>
```

(from Styles/NowStyled/app/src/main/res/layout/main.xml)

Note that the `style` attribute is part of stock XML and therefore is not in the `android` namespace, so it does not get the `android:` prefix.

The value, `@style/bigred`, points to a style resource. Style resources are values resources and can be found in the `res/values/` directory in your project, or in other resource sets (e.g., `res/values-v11/` for values resources only to be used on API Level 11 or higher). The convention is for style resources to be held in a `styles.xml` file, such as the one from the `NowStyled` project:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="bigred">
    <item name="android:textSize">30sp</item>
    <item name="android:textColor">#FFFF0000</item>
  </style>
</resources>
```

(from Styles/NowStyled/app/src/main/res/values/styles.xml)

The `<style>` element supplies the name of the style, which is what we use when referring to the style from a layout. The `<item>` children of the `<style>` element represent values of attributes to be applied to whatever the style is applied towards — in our example, our `Button` widget. So, our `Button` will have a comparatively large font (`android:textSize` set to `30sp`) and have the text appear in red (`android:textColor` set to `#FFFF0000`).

Just defining the style and applying it to the widget gives us the desired results:

Figure 173: The Styles/NowStyled sample application

Elements of Style

There are four elements to consider when applying a style:

- Where do you put the style attributes to say you want to apply a style?
- What attributes can you define via a style?
- How do you inherit from a previously-defined style (one of your own or one from Android)?
- What values can those attributes have in a style definition?

Where to Apply a Style

The `style` attribute can be applied to a widget, to only affect that widget.

The `style` attribute can be applied to a container, to affect that container. However, doing this does not automatically style its children. For example, suppose `res/layout/main.xml` looked instead like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    style="@style/bigred">
  <Button
    android:id="@+id/button"
    android:text=""
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

The resulting UI would not have the `Button` text in a big red font, despite the `style` attribute. The style only affects the container, not the contents of the container.

You can also apply a style to an activity or an application as a whole, though then it is referred to as a "theme", which will be covered a bit later in this chapter.

The Available Attributes

When styling a widget or container, you can apply any of that widget's or container's attributes in the style itself. So, if it shows up in the "XML Attributes" or "Inherited XML Attributes" portions of the Android JavaDocs, you can put it in a style.

Note that Android will ignore invalid styles. So, had we applied the `bigred` style to the `LinearLayout` as shown above, everything would run fine, just with no visible results. Despite the fact that `LinearLayout` has no `android:textSize` or `android:textColor` attribute, there is no compile-time failure nor a runtime exception.

Also, layout directives, such as `android:layout_width`, can be put in a style.

Inheriting a Style

You can also indicate that you want to inherit style attributes from another style, by specifying a `parent` attribute on the `<style>` element.

For example, take a look at this style resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
```

```
    <item name="android:background">?android:attr/activatedBackgroundIndicator</item>
  </style>
</resources>
```

(note: in some renditions of this book, you may see the `<item>` element split over two lines — this is caused by word-wrapping, as this element should be all on one line)

Here, we are indicating that we want to inherit the `Theme.Holo` style from within Android. Hence, in addition to all of our own attribute definitions, we are specifying that we want all of the attribute definitions from `Theme.Holo` as well.

In many cases, this will not be necessary. If you do not specify a parent, your attribute definitions will be blended into whatever default style is being applied to the widget or container.

That `?android:attr` looks a bit bizarre, but we will get into what that syntax means in the next section.

The Possible Values

Typically, the value that you will give those attributes in the style will be some constant, like `30sp` or `#FFFF0000`.

Sometimes, though, you want to perform a bit of indirection — you want to apply some other attribute value from the theme you are inheriting from. In that case, you will wind up using the somewhat cryptic `?android:attr/` syntax, along with a few related magic incantations.

For example, let's look again at this style resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/activatedBackgroundIndicator</item>
  </style>
</resources>
```

Here, we are indicating that the value of `android:background` is not some constant value, or even a reference to a drawable resource (e.g., `@drawable/my_background`). Instead, we are referring to the value of some other attribute — `activatedBackgroundIndicator` — from our inherited theme. Whatever the theme

defines as being the `activatedBackgroundIndicator` is what our background should be.

This portion of the Android style system is very under-documented, to the point where Google itself recommends you look at the Android source code listing the various styles to see what is possible.

This is one place where inheriting a style becomes important. In the example shown in this section, we inherited from `Theme.Holo`, because we specifically wanted the `activatedBackgroundIndicator` value from `Theme.Holo`. That value might not exist in other styles, or it might not have the value we want.

Themes: Would a Style By Any Other Name…

Themes are styles, applied to an activity or application, via an `android:theme` attribute on the `<activity>` or `<application>` element. If the theme you are applying is your own, just reference it as `@style/...`, just as you would in a `style` attribute of a widget. If the theme you are applying, though, comes from Android, typically you will use a value with `@android:style/` as the prefix, such as `@android:style/Theme.Holo.Dialog` or `@android:style/Theme.Holo.Light`.

In a theme, your focus is not so much on styling widgets, but styling the activity itself. For example, here is the definition of `@android:style/Theme.Holo.NoActionBar.Fullscreen`:

```
<!-- Variant of the default (dark) theme that has no title bar and
          fills the entire screen -->
<style name="Theme.Holo.NoActionBar.Fullscreen">
  <item name="android:windowFullscreen">true</item>
  <item name="android:windowContentOverlay">@null</item>
</style>
```

It specifies that the activity should take over the entire screen, removing the status bar on phones (`android:windowFullscreen` set to `true`). It also specifies that the "content overlay" — a layout that wraps around your activity's content view — should be set to nothing (`android:windowContentOverlay` set to `@null`), having the effect of removing the title bar.

What Happens If You Have No Theme

Most of the sample apps that we have examined so far have not defined a theme, either at the `<application>` level or the `<activity>` level. What happens here then depends upon the device that your app runs upon:

- On an Android 1.x or 2.x device, you will get `Theme` as your theme
- On an Android 3.x or 4.x device, if your `minSdkVersion` or `targetSdkVersion` is 11 or higher, you will get `Theme.Holo` as your theme; otherwise, you will stick with `Theme` as your theme
- On an Android 5.0+ device, if your `targetSdkVersion` is 14 or higher, you will get `Theme.Material` as your theme; otherwise, your app behaves as in the 3.x/4.x scenario above

As a result, your app is far from "broken", despite the lack of an explicit theme. It does mean, though, that your app will have a different look on those different Android OS levels, a look that will tend to have your app blend in more with other apps on that same device.

However, once you want to start customizing your theme, you will now run into a problem: having different themes for different OS versions. An Android 2.x device knows nothing about `Theme.Material`, for example, so you cannot simply create a custom theme based on `Theme.Material` and expect it to work. As we will see in a later chapter, the solution winds up being *versioned resources*, where you have different theme definitions for different API levels.

Of course, if your `minSdkVersion` is high enough, resource versioning is less of an issue. For example, if your `minSdkVersion` is 21, all devices that your app runs upon should know about `Theme.Material`, just as if your `minSdkVersion` were 11 or higher, all devices that your app would run on would know about `Theme.Holo`.

Android Studio's Theme Editor

Android Studio has a dedicated theme editor, which allows you to (somewhat) preview your theme and (somewhat) modify it visually.

When you open a style or theme resource, you will get a banner across the top of the XML editor, offering to open the theme in the theme editor:

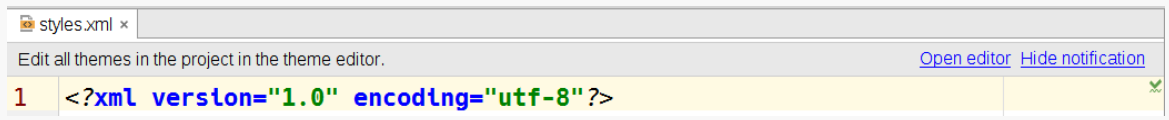


Figure 174: The Styles/NowStyled Style Resource, with Banner

Clicking the "Open editor" link in that banner will bring up the Theme Editor tab:

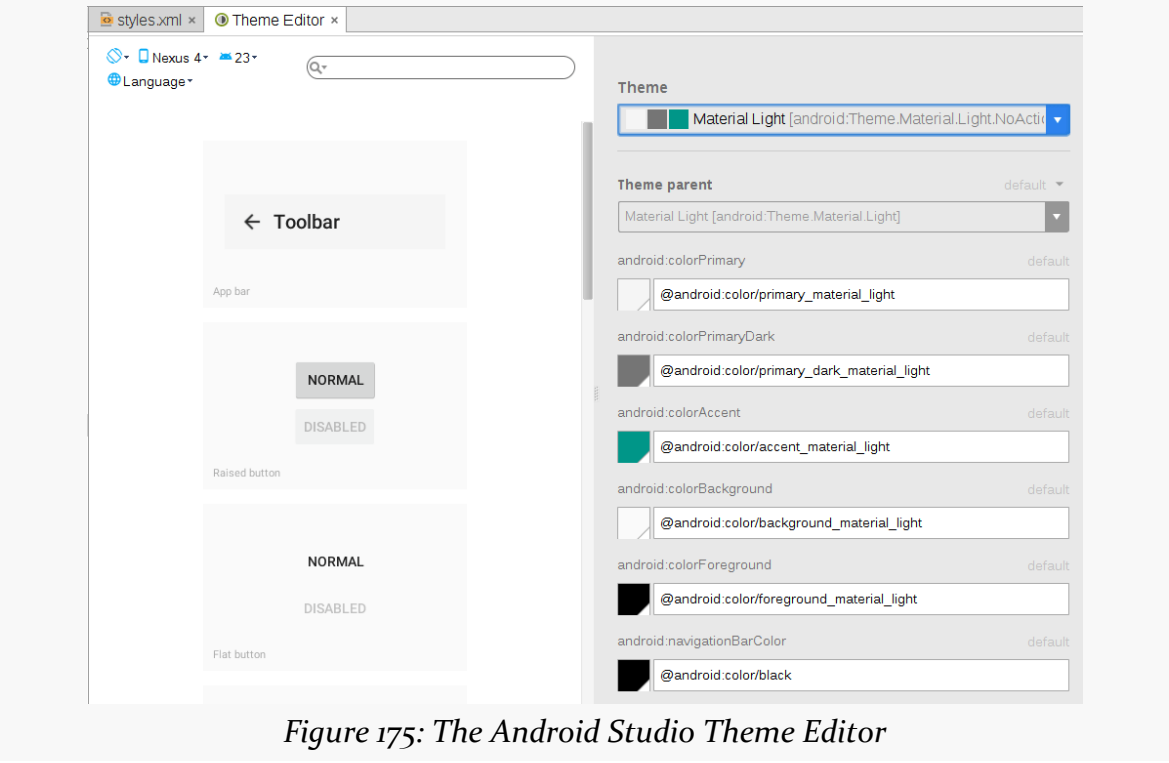


Figure 175: The Android Studio Theme Editor

If the style resource does not define a style being used as a theme – as is the case with the `NowStyled` sample app, you wind up with a pretty, albeit read-only, way of seeing how colors and settings in the theme will affect the action bar (labeled here as the "app bar"), buttons, and so forth.

If you open the Theme Editor on a style resource that is being used as a theme, you may get a preview of that custom theme:

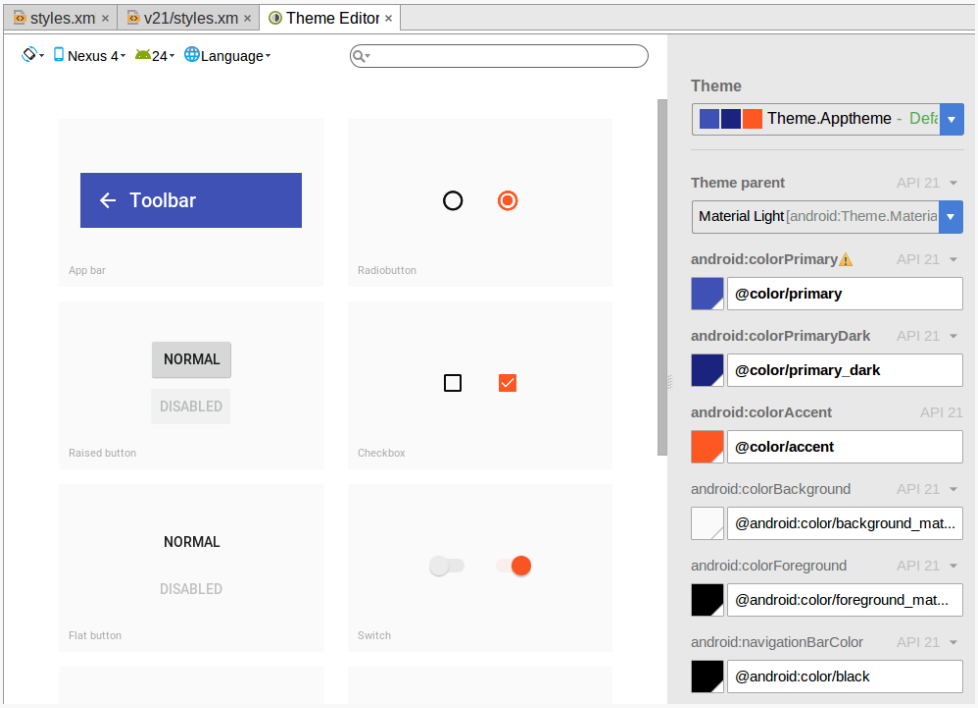


Figure 176: The Android Studio Theme Editor, For an Actual Theme

In places where you have overridden certain colors, such as the `android:colorPrimary` attribute for a `Theme.Material`-based theme, you can use a color picker to replace that color with a different value:

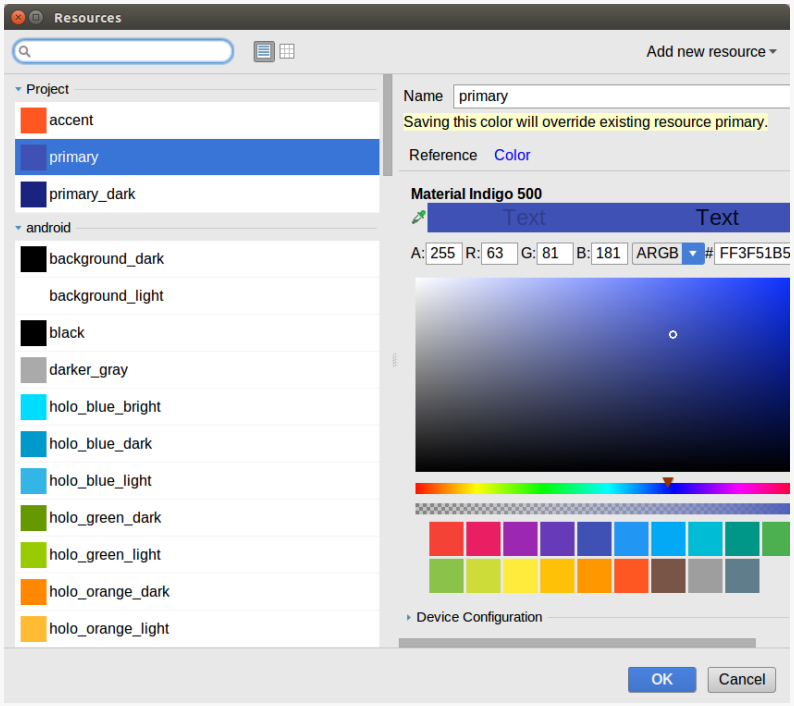
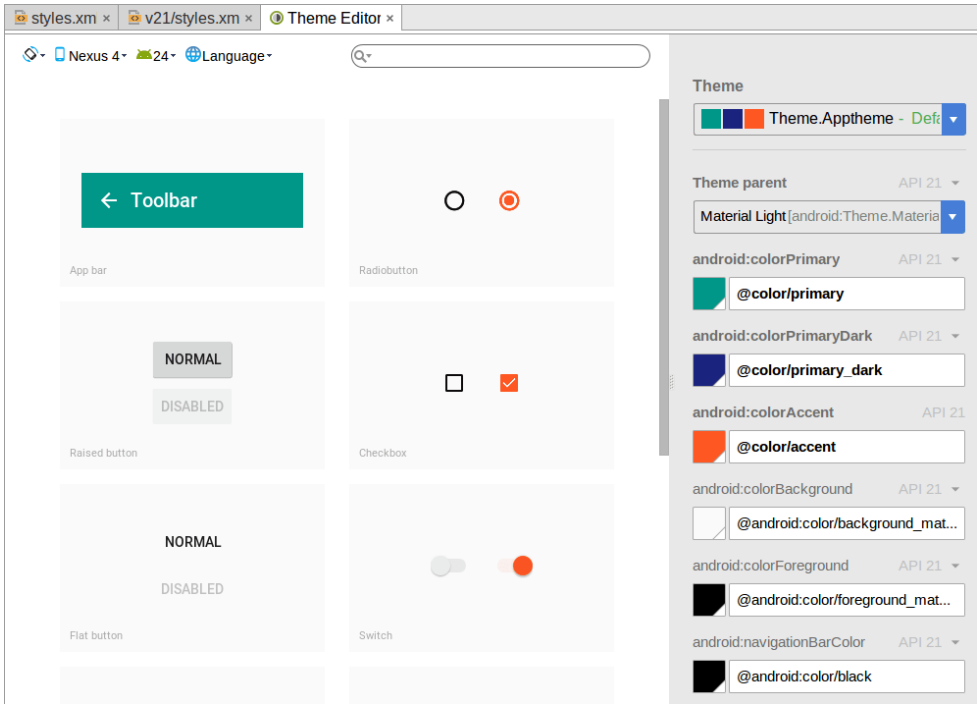


Figure 177: The Android Studio Theme Editor's Color Picker Dialog

As the dialog notes, if you change the color in the dialog, the editor will update the associated resources to match, and show you the revised value in the preview:


styles.xm ×
v21/styles.xm ×
Theme Editor ×
Nexus 4
24
Language
Toolbar
App bar
Radiobutton
NORMAL
DISABLED
Raised button
Checkbox
NORMAL
DISABLED
Flat button
Switch
Theme
Theme.Apptheme - Defa
Theme parent
API 21
Material Light [android:Theme.Materia
android:colorPrimary
API 21
@color/primary
android:colorPrimaryDark
API 21
@color/primary_dark
android:colorAccent
API 21
@color/accent
android:colorBackground
API 21
@android:color/background_mat...
android:colorForeground
API 21
@android:color/foreground_mat...
android:navigationBarColor
API 21
@android:color/black

Dependencies

While you are writing some code for an app, the vast majority of the code that *is* the app comes from other developers. Some might be teammates on your development team, but far more comes from outsiders: Google and other Android developers.

Some of this you have seen already. You did not write `Activity`, `TextView`, and similar classes. Instead, they came from the Android SDK, written (primarily) by Google.

Beyond the Android SDK, though, there are thousands of libraries that developers have access to, including many from Google itself. We add these as dependencies in our projects, to use their code alongside ours.

What's a Dependency?

Roughly speaking, the code and assets that make up an app come from three sources.

The source that you tend to focus on personally is the code that you and people that you know are writing for this app.

There is the source that comes from your `compileSdkVersion`, representing the Android SDK that you are linking to.

Everything else, generally speaking, is a dependency.

From a pure technical standpoint, dependencies are listed in `build.gradle` files in `dependencies` closures.

Dependency Scopes

One `dependencies` closure appears in the project-level `build.gradle` file, inside of a `buildscript` closure:

```
// Top-level build file where you can add configuration options common to all
sub-projects/modules.

buildscript {

    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0'


        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Those list places where Gradle plugins come from. You are always depending upon the Android Gradle Plugin, and some other developers publish Gradle plugins that you may elect to use in the future.

However, the `dependencies` closure that we tend to think about the most is the one in our module's `build.gradle` file, such as `app/build.gradle`, such as this closure:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    testImplementation 'junit:junit:4.12'
```

```
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

Here, there are three types of statements:

- `implementation` says "here is a dependency that I want to use for everything"
- `androidTestImplementation` says "here is a dependency that I want to use for instrumentation tests"
- `testImplementation` says "here is a dependency that I want to use for unit testing"

There are other possibilities, and we will be exploring those and the testing-related statements later in this book. For now, we will focus on `implementation` statements.

Depending on a Local JAR

The first of those `implementation` statements is:

```
    implementation fileTree(dir: 'libs', include: ['*.jar'])
```

This pulls in any JAR files that happen to be in the `libs/` directory of this module.

JARs, as you probably know, are libraries containing Java code, as created by standard Java build tools (`javac`, `jar`, etc.). For the first decade-plus of Java's existence, we distributed reusable bits of code in the form of JAR files. You would download a JAR from a Web site, drop it into your project, and through something like this `implementation` statement, say that your project should use the JAR.

In Android, the contents of these JARs are packaged into your APK. And, whatever `public` classes happen to be in those JARs are available to you at compile time.

However, in general, using plain JARs nowadays is considered to be a bad idea. There is no information in a JAR about:

- What version of the library the JAR represents — while this could be part of the filename, files can be renamed far too easily
- What other libraries this JAR requires — at best, you find that out from documentation, then need to hunt down those JARs and see what *they* require, and so on

Instead, nowadays, you should try to use artifacts, rather than bare JAR files.

What's an Artifact?

An artifact is usually represented in the form of two files:

- The actual content, such as a JAR
- A metadata file, paired with the JAR, that has information about "transitive dependencies" (i.e., the other artifacts that this artifact depends upon)

Artifacts and Repositories

Artifacts are housed in artifact repositories. Those repositories not only contain the artifacts, but they organize the artifacts for easy access. This includes organizing them by version, so you can request a specific version of an artifact and get it, instead of an older (or possibly newer) version of that same artifact.

Some artifact repositories are public. A typical Android project will use two such repositories:

- JCenter, a popular place for open source artifacts
- Google's repository, for things like the Android Gradle Plugin

There are other general-purpose artifact repositories, such as Maven Central or jitpack.io. There are specialty repositories, such as the one used by the author of this book for his artifacts. And there can be private repositories, such as ones used by organizations for their own private artifacts, used by their private projects.

Major Library Families from Google

Beyond what strictly is part of the Android SDK, Google publishes a lot of libraries that developers can opt into by declaring them as dependencies.

These include:

- The Android Support Library, which is the original and largest collection of support code that Google publishes, and which will be used frequently throughout the remainder of this book

- The `ConstraintLayout` library, which we will explore in an upcoming chapter
- The Architecture Components, which are covered in *Android's Architecture Components*, a companion volume to this book by the same author
- The data binding library, which we will explore in an upcoming chapter
- The testing support library, which contains several classes that we will examine in the chapters on app testing
- The Play Services SDK, which will we cover briefly later in the book

Requesting Dependencies

With all that as background, let's explore a bit more about how those `implementation` statements are working and how you can add your own for other dependencies that you may want to use.

Find What You Need

First, you need to identify the dependencies that address whatever problems you need to solve.

Many dependencies will be covered in this book. Yet more are covered in the aforementioned *Android's Architecture Components* book. A few more are in the developer documentation.

The biggest catalog of open source dependencies is the Android Arsenal. Here you can browse and search for dependencies. Each listing will contain links to where you can find out more about that particular library, typically in the form of a GitHub repository.

Configure the Repositories

If you are using one of Google's libraries, a project created in Android Studio 3.0+ should be set up with the proper artifact repository already. Your project also comes pre-configured to pull from JCenter, where many open source libraries are from. As a result, for the vast majority of libraries, you do not need to configure an additional artifact repository.

But, sometimes you do.

For example, the author of this book has published a number of open source libraries, collectively referred to as the CommonsWare Android Components (CWAC). They are published in the author's own artifact repository. When you visit the GitHub repository for one of these libraries, you will see that in the installation instructions, it has a sample `repositories` closure:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}
```

You can do one of two things:

1. Add that `maven {}` closure to the `repositories` closure in the `allprojects` closure in your top-level `build.gradle` file
2. Add the entire sample `repositories` closure to your module's `build.gradle` file

Either of these will make this artifact repository available to you for that module. The first approach makes it available for *all* modules in your app, which may become useful if later you create a library module.

So, when you review the documentation for a dependency, it should indicate what artifact repository to use, and you need to ensure that you are set up to use that repository.

Identify the Dependencies and Versions

Each artifact has an identifier made up of three pieces: a group ID, an artifact ID within the group, and a version number.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

The first `implementation` statement shown above pulls in bare JARs located in your project's filesystem. The remaining statements in this `dependencies` closure

represent artifacts. The three ones that begin with `com.android` are from Google, while the one that starts with `junit` is an open source testing library called JUnit.

You will see that all four of those artifacts have the three-part identifier:

- the group ID (e.g., `com.android.support.constraint`)
- the artifact ID (e.g., `constrant-layout`)
- the version number (e.g., `1.0.2`)

Technically, the version number can contain wildcards. For example, while `1.0.2` indicates a specific version, `1.0.+` says "pick the latest among all versions that start with `1.0`". This is convenient for getting patches, but it means that on some random day, all of a sudden, you are using a new version of the library, and that might cause problems. Typically, we do not use wildcards, but instead just keep tabs on when the artifacts get new versions. Android Studio will help with this, highlighting artifacts that have newer versions available.

Add the Dependencies

Then, you just need to add the appropriate `implementation` lines for whatever dependencies that you wish to add. So, for example, if you wanted to add the CWAC-NetSecurity library to your project, in addition to adding that CWAC repository to your repositories list, you could add:

```
implementation 'com.commonsware.cwac:netsecurity:0.4.4'
```

to your `dependencies` closure.

Note that many libraries showing sample code for adding them to your `build.gradle` file will show a slightly different syntax:

```
compile 'com.commonsware.cwac:netsecurity:0.4.4'
```

That is because Android Studio 3.0 and Gradle 4.1 switched to a new syntax for specifying dependencies. `compile` was replaced by `implementation`. If you use `compile`, your Gradle build script will still work... for now. Eventually, support for `compile` will be dropped, and so you should aim to use `implementation` instead of `compile` going forward.

Also note that some of the sample projects in this book will use `compile`, as the book slowly adjusts to the new syntax.

We will see lots of `dependencies` closures throughout the rest of this book, showing different artifacts that we can depend upon.

The Android Support Library

The Android Support Library is a series of artifacts distributed by Google, containing general-purpose classes (in JARs and Android library projects) that are not part of the Android SDK, but are available to Android developers.

What's In There?

You can roughly divide the contents of the Android Support Library into two major areas:

1. "Backports" of capabilities added to newer versions of Android and the Android SDK, so they can be used on older devices as well. By using the backported classes, you can get the same abilities on a wider range of devices than you could if you only used the classes in the Android SDK.
2. New widgets, containers, or other classes that are not going to be in the Android SDK (for ill-defined reasons) but that Google wishes to make available for Android developers.

Example artifacts of the Android Support Library include:

- `appcompat-v7`, which is a backport of the action bar, a concept that we will discuss in an upcoming chapter
- `recyclerview-v7`, which is the home of the `RecyclerView` widget that serves as an alternative to `ListView` and `GridView`
- `support-compat`, with compatibility classes to make it easier to support both old versions of Android and new ones
- `support-core-ui`, offering some widgets and containers, such as `ViewPager`
- `support-fragment`, providing a backport of fragments

About the Names

What this book refers to as the "Android Support Library" has many names.

It was originally referred to as the Android Compatibility Library, at a time when it only contained backports. Once Google started adding in things that were not strictly related to "compatibility", they started changing the name to try to be more

generic. Right now, "Android Support" seems to be fairly consistent, either used standalone or in the form of "Android Support Library" or "Android Support Libraries".

About the -v Suffixes

Some artifact names have version suffixes, like `recyclerview-v7`. Others do not, such as `support-fragment`. The `-v7` naming convention used to indicate the `minSdkVersion` supported by that artifact. Originally, `recyclerview-v7` worked back to API Level 7 and did not support older versions of Android than that.

However, that was a fairly inflexible system. Not only has Google abandoned it for newer artifacts, but the actual value no longer has meaning. In particular, the entire Android Support Library only supports back to API Level 15 at this point, with a few exceptions that do not go back even that far.

Getting It

Google is now distributing its libraries via their own Maven-style artifact repository. With Android Studio 3.0, this repository is what you get from that `google()` statement in the `repositories` closures.

For most libraries for most versions that you are likely to encounter, the `google()` repository will have what you need. You can visit the Google Maven Repository site to see what artifacts and versions are served from it.

Originally, these libraries were distributed via the Android Support Repository, one that you would have on your own hard drive, downloaded via the SDK Manager.

Choosing a Version

Typically, and historically, developers would take a three-tier approach towards the version of these libraries:

- When starting a new project, developers would use the then-current edition of the libraries, along with the latest `compileSdkVersion`
- When a new patch release to the libraries was released — for example, from `25.3.0` to `25.3.1` — developers would update to that version, as usually the patches fix bugs

- When it came time for the project to move to a new `compileSdkVersion`, developers would move to the matching major version of the libraries (e.g., when moving to `compileSdkVersion 26`, switch to version `26.1.0` of the libraries)

That is still a fine pattern... if your `minSdkVersion` is 15 or higher. For many developers, that will be the case.

However, if you are aiming to still support Android 2.x devices, you have a problem: starting with the v26 edition of the Support Library, the oldest version of Android supported by the libraries is Android 4.0.3, API Level 15. This means that you will need to stick with `compileSdkVersion 25` and the v25 edition of the Support Library until such time as you either:

- Remove all ties to the Support Library, or
- Raise your `minSdkVersion` to 15 or higher

Attaching It To Your Project

You can add references to the Android Support Library's libraries — whether those libraries are simple JARs or Android library projects — via a few lines in your `dependencies` closure, referencing the artifacts from the Android Support Repository.

Here are the `implementation` statements for many artifacts in the Android Support Library:

```
implementation 'com.android.support:appcompat-v7:27.0.2'
implementation 'com.android.support:cardview-v7:27.0.2'
implementation 'com.android.support:design:27.0.2'
implementation 'com.android.support:exifinterface:27.0.2'
implementation 'com.android.support:gridlayout-v7:27.0.2'
implementation 'com.android.support:leanback-v17:27.0.2'
implementation 'com.android.support:mediarouter-v7:27.0.2'
implementation 'com.android.support:palette-v7:27.0.2'
implementation 'com.android.support:percent:27.0.2'
implementation 'com.android.support:recyclerview-v7:27.0.2'
implementation 'com.android.support:support-annotations:27.0.2'
```

Also, while you *could* add all of these to your project, that is not necessary. Only attach dependencies for libraries that you are actually using. Having unused libraries

in your project just increases your APK size for no good reason. Hence, most projects will have only a subset of the aforementioned lines.

Note that, in general, when using the Android Support libraries, you should set your `compileSdkVersion` and `targetSdkVersion` to be the same as the major version of the library. So, for a `26.1.0` version of the library, your `compileSdkVersion` should be `26`.

Tutorial #6 - Adding a Library

We will want to use some third-party libraries in our project, to ease development of the app:

- the Android Support library, specifically its `android-support-v13` JAR
- greenrobot's EventBus, for communication between various pieces of our app
- Google's Gson parser of JSON data
- Square's Retrofit, including its Gson-based converter code, for retrieving JSON data from Web services,
- Square's OkHttp, for general HTTP requests, like downloading a ZIP archive
- the CWAC-Security library, by the author of this book, which contains some code for securely unpacking a ZIP archive
- an implementation of tabs

Right now, we will just focus on arranging for our project to be able to use the libraries. Later in the book, we will actually put the libraries to use.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Getting Rid of Existing Cruft

If you look at the `app/build.gradle` file, you will see that we already have some dependencies:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
```

```
    androidTestImplementation('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    testImplementation 'junit:junit:4.12'
}
```

(from EmPubLite-AndroidStudio/T5-Layout/EmPubLite/app/build.gradle)

These dependencies are the sort of thing that gets added automatically to your project by the new-project wizard. While you imported your starter project, rather than creating it from scratch, the starter project has these dependencies because *it* was created from the new-project wizard.

However, we are not going to use any of those dependencies:

- The `implementation fileTree()` line pulls in bare JAR files from the `libs/` directory, and we will not be using bare JAR files
- The other statements pull in dependencies for testing, and while testing is a fine thing to do, these tutorials do not have you write any test cases

However, not only does the new-project wizard generate dependencies like these for us, but it also code-generates some do-nothing test code that depends upon these dependencies. So, we will leave those two test dependencies alone, as it is simpler to ignore them than it is to clean that part up.

But go ahead and delete the `implementation fileTree` statement, leaving you with

```
dependencies {

androidTestImplementation('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    testImplementation 'junit:junit:4.12'
}
```

You may get a yellow banner at the top of the editor, indicating that a “project sync” is requested. Ignore that for the moment, as we will be making more changes to this file.

Step #2: Requesting New Dependencies

Many of the dependencies we are going to set up now are available from JCenter, and our project is already set up to pull from there. However, the CWAC-Security library is not, and so we will need to teach Gradle how to find that library.

To do that, add the following code to your `app/build.gradle` file, above the `dependencies` closure:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}
```

(from EmPubLite-AndroidStudio/T6-Library/EmPubLite/app/build.gradle)

Then, add seven more lines to the `dependencies` closure, identifying the libraries that we need:

```
dependencies {
    androidTestImplementation('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    testImplementation 'junit:junit:4.12'
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation 'com.google.code.gson:gson:2.8.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
    implementation 'com.squareup.okhttp3:okhttp:3.4.1'
    implementation 'com.commonsware.cwac:security:0.8.0'
    implementation 'com.android.support:support-v13:25.3.0'
    implementation 'io.karim:materialtabs:2.0.5'
}
```

(from EmPubLite-AndroidStudio/T6-Library/EmPubLite/app/build.gradle)

At this point, your `app/build.gradle` file should look something like:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'
    defaultConfig {
        applicationId "com.commonsware.empublite"
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

repositories {
```

```
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    androidTestImplementation('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    testImplementation 'junit:junit:4.12'
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation 'com.google.code.gson:gson:2.8.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
    implementation 'com.squareup.okhttp3:okhttp:3.4.1'
    implementation 'com.commonsware.cwac:security:0.8.0'
    implementation 'com.android.support:support-v13:25.3.0'
    implementation 'io.karim:materialtabs:2.0.5'
}
```

(from EmPubLite-AndroidStudio/T6-Library/EmPubLite/app/build.gradle)

If that "project sync" yellow banner is at the top of the editor, click the "Sync Now" link in that banner to synchronize the `*.iml` files with the changes you made to this `build.gradle` file. If it is not there, choose Tools > Android > Sync Project with Gradle Files to force that resync.

In Our Next Episode…

… we will configure the action bar on our tutorial project.

Introducing ConstraintLayout

In 2016, Google released a new container class, `ConstraintLayout`, that it hopes will become popular among Android developers as an alternative to `LinearLayout` and `RelativeLayout`. Certainly, Google is going to "pull out all the stops" to convince developers to use `ConstraintLayout`, such as having it be used in many of the activity templates employed by Android Studio's new-activity wizard.

`ConstraintLayout` is useful, but it is not required for Android app development, any more than `LinearLayout` and `RelativeLayout` are. And, since `ConstraintLayout` is a library, it adds ~100KB to the size of your Android app. Whether it is worth that extra space is for you to decide.

Why Another Container?

`LinearLayout`, `RelativeLayout`, and (to a lesser extent) `TableLayout` have served as the backbone of most Android apps. Previous attempts to provide a new foundation container class — such as `GridLayout` – have not proven to be particularly popular.

So, why did Google bother creating `ConstraintLayout`?

Drag-and-Drop GUI Builders

Google would like everyone to use Android Studio, and in particular for everyone to use Android Studio's drag-and-drop GUI builder.

How well a drag-and-drop GUI builder works depends a lot on how the rules for laying out a UI get defined. With drag-and-drop gestures, the developer is only providing you with X/Y coordinates of a widget, based on where the developer

releases the mouse button and completes the drop. It is up to the GUI builder to determine what that really means in terms of layout rules.

With `LinearLayout`, adding a widget is fairly easy:

- If the developer drops the widget between two existing children of a `LinearLayout`, put the new widget in between the existing ones
- Otherwise, add the widget to the end of the `LinearLayout` where the developer dropped it

`TableLayout` is a bit more involved but still not that bad, as we have decades of experience of working with spreadsheets to know about inserting rows and columns into a grid-like structure.

`RelativeLayout`, though, was difficult for a GUI builder to handle. Often, the Android Studio GUI builder (and its predecessor in Eclipse) would misinterpret the developers wishes. Sometimes, the rules the developer wanted to express were simply unavailable through pure drag-and-drop operations. As a result, developers had to dive into the XML to get anything done. Being able to read layout XML is important – otherwise, books like this would be unusable. However, forcing developers to write the XML defeated the purpose of the GUI builder.

`ConstraintLayout` was created with GUI building in mind, to make it a bit easier to infer the right rules based upon where the developer happens to drop a widget.

Performance

`RelativeLayout`, `LinearLayout` with `android:layout_weight`, and `TableLayout` with `android:stretchColumns`/`android:shrinkColumns`, all require two passes over their children to determine final sizes and positions. For example, with a weighted `LinearLayout`, you need to make one pass to calculate the directly-expressed sizes (e.g., `android:layout_height`), followed by a second pass to allocate the remaining space according to the weight.

This gets exacerbated by the fact that changing the details of a widget often causes the sizes to have to be recomputed. Suppose that you change the text in a `TextView` with `android:layout_width="wrap_content"`. Changing the text changes the horizontal space taken up by that text. So, the `TextView` winds up telling its container “hey, please recompute the sizes and positions”, as the larger `TextView` might now cause shifts in other children of the container. Depending on how the

container itself is sized, it might need to tell *its* container to recompute the sizes and positions.

Cascading upward to have parents re-size/re-position their children gets very expensive for deep hierarchies, where we have containers holding containers holding containers holding containers and so on. One change in text of a `TextView` might cause that whole hierarchy to go through re-size/re-position work.

All else being equal:

- Deeper view hierarchies are slower to render than are shallower ones
- View hierarchies where parents need two passes to size and position their children are slower to render than are hierarchies where only one pass is needed

`ConstraintLayout` is being designed with performance in mind, trying to eliminate as many two-pass scenarios as possible and by trying to eliminate the need for deeply-nested view hierarchies. Right now, `ConstraintLayout` performance is not that great. However, it is a focus area and should improve over time.

Comparing with the Classics

Stylistically, `ConstraintLayout` most closely resembles `RelativeLayout`. As with `RelativeLayout`, you can anchor widgets to other widgets inside the `ConstraintLayout` or to the boundaries of the `ConstraintLayout` itself.

Many structures that can be implemented using `LinearLayout` but not `RelativeLayout` — such as allocating widget sizes based on weights – can be handled by `ConstraintLayout`. `ConstraintLayout` is designed to handle such conditional sizing without requiring two passes through its children to determine those sizes.

However, `TableLayout` remains distinct. `ConstraintLayout` does not have the notion of columns, let alone sizing those columns based upon their contents and layout rules.

Getting ConstraintLayout

To use `ConstraintLayout`, you do not need to do anything special with your SDK Manager. All that you need to do is to request some version of the

`com.android.support.constraint:constraint-layout` artifact in your `dependencies` closure of your module's `build.gradle` file:

```
dependencies {
  implementation 'com.android.support:support-v13:25.3.1'
  implementation 'com.android.support.constraint:constraint-layout:1.0.2'
}
```

(from Containers/Sampler/app/build.gradle)

For Android Studio 2.2+ users, getting this dependency is similar to the rest of the support libraries. You download the right bits to your development machine using the Android Studio SDK Manager, and the Android Plugin for Gradle then knows where to look for it. However, for inexplicable reasons, `ConstraintLayout` requires downloading two additional SDK Manager items:

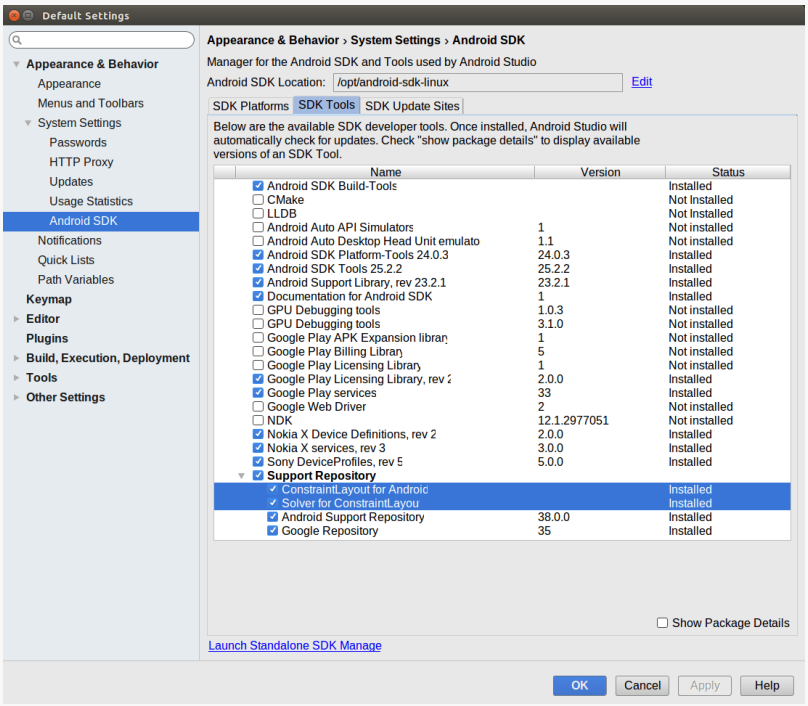


Figure 178: SDK Manager, With ConstraintLayout Items Highlighted

Getting `ConstraintLayout` outside of Android Studio is undocumented and complicated at the present time.

Using Widgets and Containers from Libraries

When you create layout resources using widgets and containers that are part of the main Android SDK, you:

- Use `android:` prefixes for nearly every attribute, and
- Use a bare class name (e.g., `Button`), instead of a fully-qualified class name (e.g., `android.widget.Button`), for nearly every element name

When you use widgets and containers from libraries — including those from the Android Support libraries — you:

- Use a mix of an `android:` and `app:` prefixes for attributes, where the `app:` ones are for attributes that are not part of the main Android SDK
- Use a fully-qualified class name for the element names (e.g., `android.support.constraint.ConstraintLayout`)

That `app:` prefix requires another XML namespace declaration: `xmlns:app="http://schemas.android.com/apk/res-auto"`. Usually, this will be added to your layout resource automatically or via a quick-fix.

When we start reviewing the XML for `ConstraintLayout`, you will see both of these changes come into effect.

Using a ConstraintLayout

Back in the chapter on the classic container classes, we reviewed several layouts from the `Containers/Sampler` sample project. That project also happens to use `ConstraintLayout`, and it has several layouts that demonstrate how `ConstraintLayout` can be used in place of the classic containers. They also serve to illustrate how to use `ConstraintLayout` in general.

Basic Anchoring, Single Axis

In the chapter on classic containers, we had the “bottom-then-top” scenario, with a small `Button` on the bottom, underneath another `Button` that took up all the remaining space:

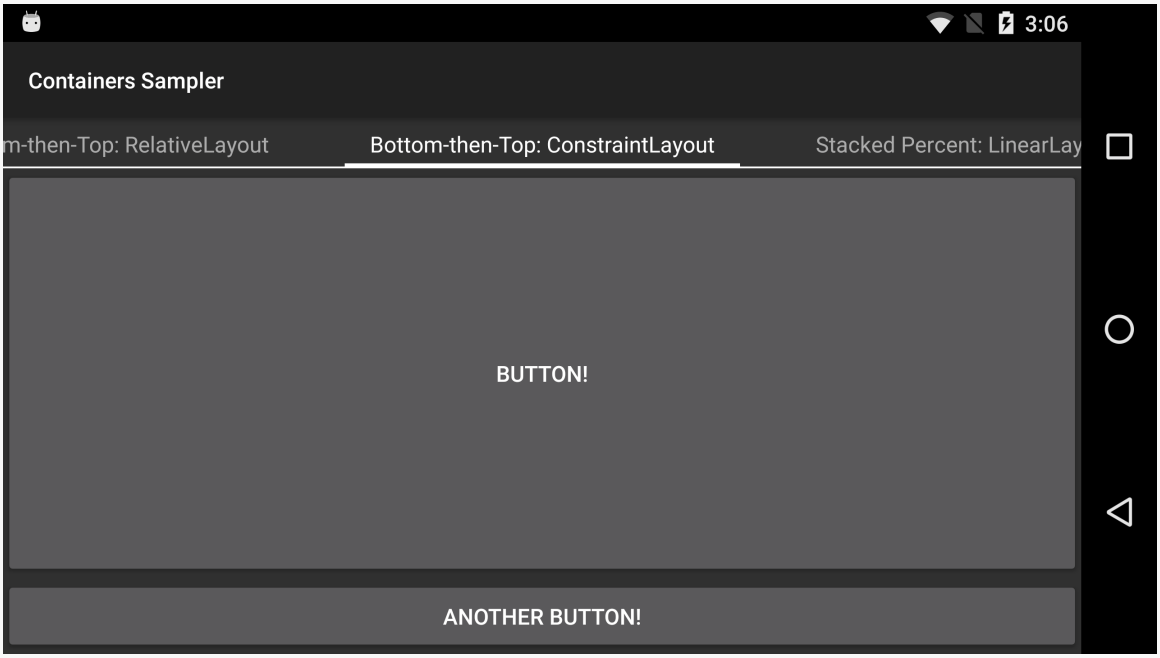


Figure 179: Bottom-then-Top Layout, Using ConstraintLayout

In addition to being able to implement this using `LinearLayout` and using `RelativeLayout`, you can implement this using `ConstraintLayout`. Here, we are setting the sizes and locations of two widgets, but only focusing on a single axis at the moment: the vertical axis.

The XML

The `ConstraintLayout` approach resembles that of the `RelativeLayout` implementation. We use layout rules to tie the bottom `Button` to the bottom of the `ConstraintLayout` and tie the top `Button` between the top of the `ConstraintLayout` and the top of the bottom `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:text="@string/button"
    app:layout_constraintBottom_toTopOf="@+id/another_button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
```

```
    app:layout_constraintTop_toTopOf="parent" />

  <Button
    android:id="@id/another_button"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="@string/another_button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" />
</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/bottom_then_top_cl.xml)

The bottom `Button` has three `ConstraintLayout` rules, such as `app:layout_constraintBottom_toBottomOf="parent"`. `ConstraintLayout` rules are verbose and a bit odd-looking. It may help to unpack this into pieces:

- `app:` is because `ConstraintLayout` comes from a library, not the Android framework itself
- `layout_` is the standard prefix for rules based upon a widget's container, no different than what we saw with the classic containers
- `constraint` is used in all the `ConstraintLayout`-specific rule attribute names
- `Bottom` indicates what side of the widget we are looking to anchor (in this case, the bottom of the widget)
- `toBottomOf` indicates what side of the target (`parent`) we want to anchor to (in this case, the bottom of the `ConstraintLayout`)

The value associated with the rule attribute is either a widget ID of another child of the same `ConstraintLayout`, or `parent` to indicate the `ConstraintLayout` itself. So, our `Button` is anchored to the bottom of the `ConstraintLayout`, with its natural `wrap_content` height.

The three rules on the bottom `Button` are:

- `app:layout_constraintBottom_toBottomOf="parent"`, to tie the bottom of the `Button` to the bottom of the `ConstraintLayout`
- `app:layout_constraintEnd_toEndOf="parent"`, to tie the end of the `Button` to the end of the `ConstraintLayout`
- `app:layout_constraintStart_toStartOf="parent"`, to tie the start of the `Button` to the start of the `ConstraintLayout`

Our top `Button` has four `ConstraintLayout` rule attributes:

- `app:layout_constraintBottom_toTopOf="@+id/another_button"` anchors the bottom of this `Button` to the top of the bottom `Button`, based upon the ID of the bottom `Button`
- `app:layout_constraintTop_toTopOf="parent"` anchors the top of the top `Button` to the top of the `ConstraintLayout`
- `app:layout_constraintEnd_toEndOf="parent"` anchors the end of the top `Button` to the end of the `ConstraintLayout`
- `app:layout_constraintStart_toStartOf="parent"` anchors the start of the top `Button` to the start of the `ConstraintLayout`

Sizes of widgets also play a role here. The height of the bottom `Button` is `wrap_content`, so it will be its natural height. The height of the top `Button` is `0dp`, indicating that we want to stretch it between anchor points established by rules. The width of both buttons is also `0dp`, indicating that we want those buttons to be stretched between anchor points along the horizontal axis.

Now, you might wonder why we do not just set the width of the buttons to `match_parent` and skip the "start" and "end" rules. In an ideal world, this is what we would do. The developers of `ConstraintLayout` have other ideas, and those ideas include making `match_parent` useless. If you try to use `match_parent`, the actual width that you get will vary by circumstance and rarely actually fill the available space. So, we are stuck with the `0dp`-and-anchor-rules approach.

The Android Studio Graphical Layout Editor

As with other containers, you can create a new layout resource with a `ConstraintLayout` as the root, by right-clicking over a `layout` resource directory, choosing New > "Layout resource file" from the context menu, and typing in `android.support.constraint.ConstraintLayout` for the root element. Fortunately, auto-complete on the "Root element" field allows you to just start typing `ConstraintLayout`, then choose the fully-qualified class name from the drop-down list.

If you drag a widget into the `ConstraintLayout` and drop it in an arbitrary spot, what you get at design time will be different than what you get when you run the app. In the graphical layout editor, the `Button` shows up where you drop it:

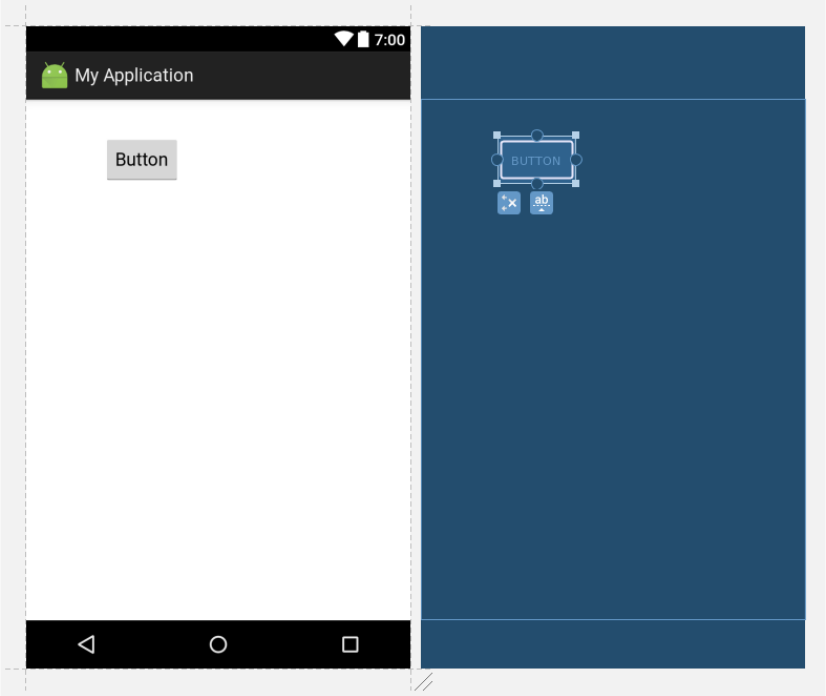


Figure 180: ConstraintLayout, With Dragged-In Button

However, if you look at the XML that was generated, you will see that the `Button` has no constraints. It *does* have a pair of attributes with the `tools:` prefix: `tools:layout_editor_absoluteX` and `tools:layout_editor_absoluteY`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/button9"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Button"
    tools:layout_editor_absoluteX="77dp"
    tools:layout_editor_absoluteY="36dp" />
</android.support.constraint.ConstraintLayout>
```

Attributes in the `tools:` namespace are suggestions to the development tools and have no impact on the behavior of your app when it runs. In this case, Android

Studio remembers the upper-left corner of where you dropped the `Button`. But, as a warning on the `Button` in the layout editor will tell you, a `Button` without constraints will wind up at coordinate `(0,0)` at runtime (basically, upper-left for LTR languages and upper-right for RTL languages).

Dragging in a widget is insufficient. You also need to use the graphical layout editor to define the constraints.

If you click on a widget that you dragged into the `ConstraintLayout`, the blueprint view will show squares on the corners and circles centered on the edges, plus a couple of bubbles beneath it:

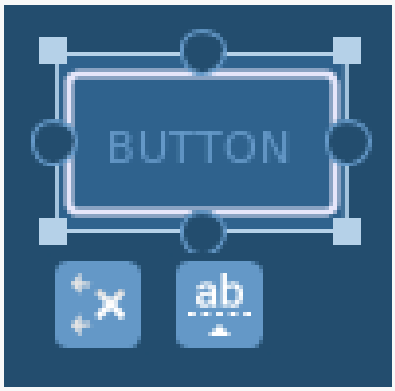


Figure 181: Blueprint View, Showing Annotated Widget

The squares are resize handles. Most likely, you have seen this pattern before, whether in IDEs, drawing tools, or other programs. You would use this resizing approach if you wanted a fixed size for the widget. Later switching to using dimension resources, rather than hard-coded values, for the size values would be a good idea. You can also change the width and height through the Attributes pane, as with widgets inside of other sorts of containers.

The circles are more important, as they allow you to define the constraints, by dragging a circle to some anchor point:

Figure 182: Blueprint View, Showing Constraint Being Created

To create the bottom `Button` from the layout shown in the previous section, you would set its width to be `0dp`, then use the circles to establish the three constraints, on the bottom, start, and end:

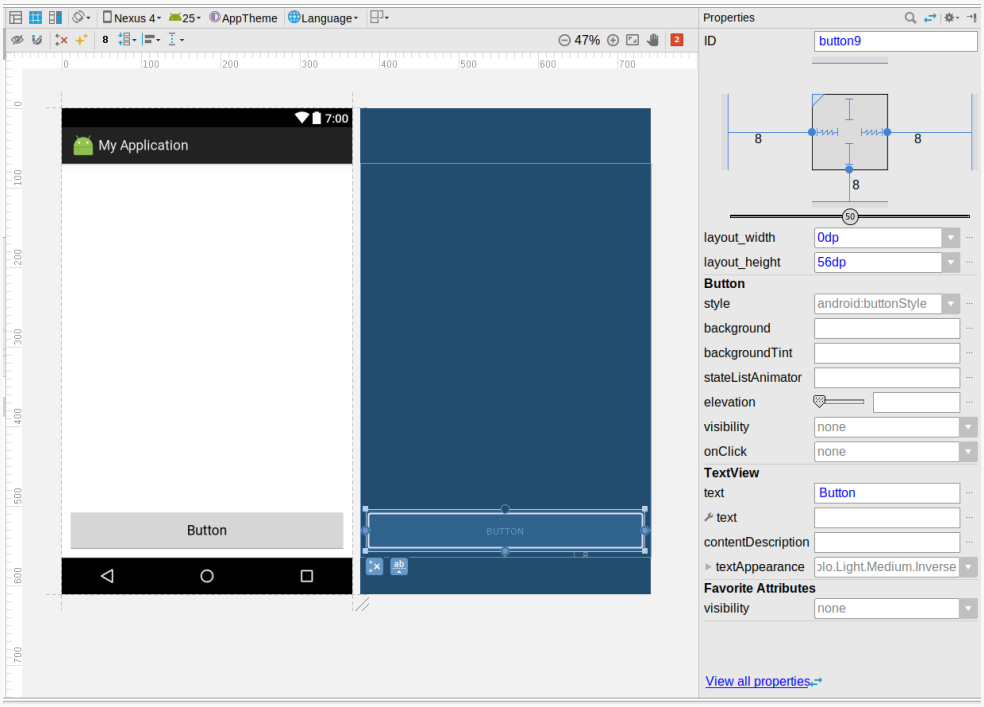


Figure 183: Blueprint View, Showing Three Constraints on Bottom Button

To create the top `Button`, you would drag another `Button` into the `ConstraintLayout`, set its width and height to `0dp`, then set up the four constraints, one on each side. In the case of the constraint starting from the bottom-edge circle of the top `Button`, you would drag it to the top edge of the bottom `Button` to tie those widgets together:

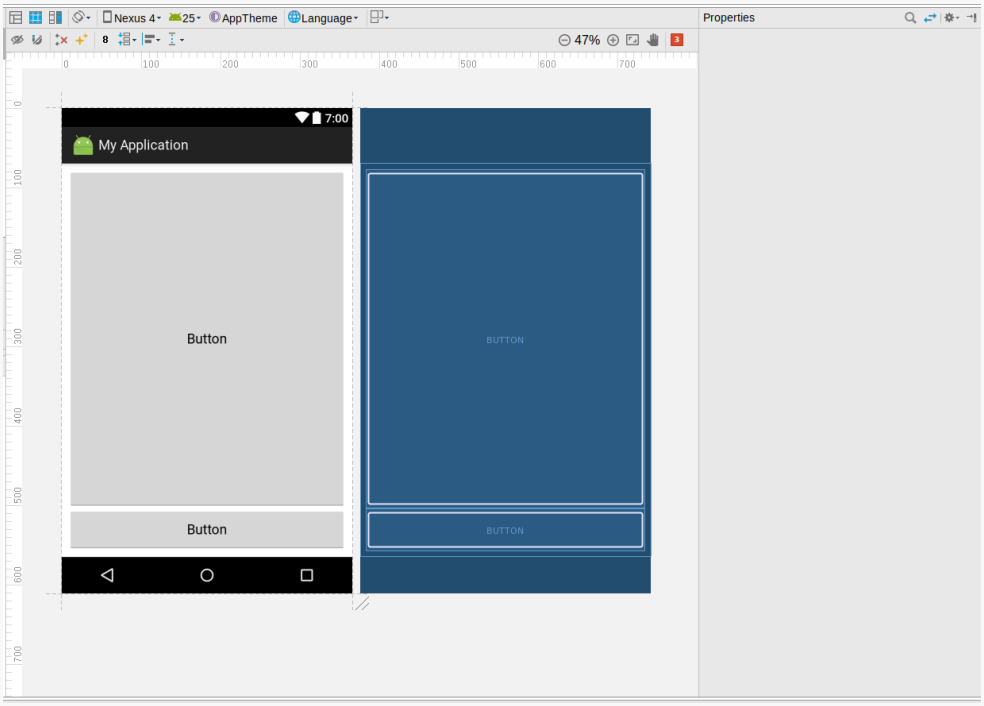


Figure 184: Preview and Blueprint View, Showing Two Buttons

When you click on a widget inside a `ConstraintLayout`, the Attributes pane has a strange-looking control above the regular properties:

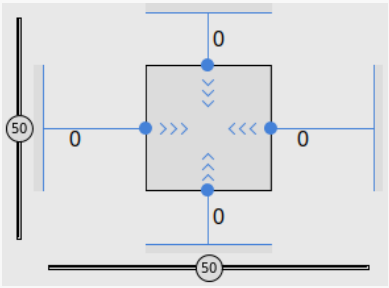


Figure 185: Constraint Configuration Thingy

This allows you to manage some aspects of the constraints. For example, the four 0 values shown in that image turn into comboboxes when clicked, and their values form the margins on that side of the widget.

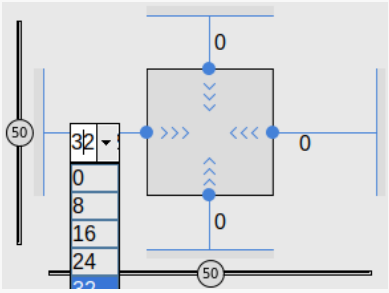


Figure 186: Constraint Configuration Thingy, Showing Margin Combobox

The four sets of chevrons indicate that the sizing rules are `wrap_content` for both the `layout_width` and `layout_height`. Clicking one of the chevrons will toggle between three states for that axis:

- `wrap_content`, indicated by chevrons
- a fixed width, indicated by a sizing bar

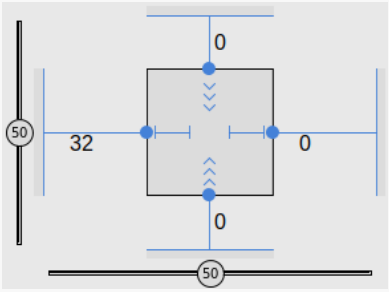


Figure 187: Constraint Configuration Thingy, Showing Fixed Horizontal Sizing

- `0dp`, for stretching the size to the available space, indicated by a sawtooth line

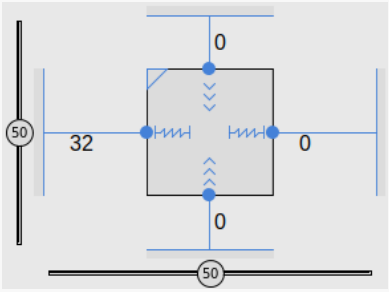


Figure 188: Constraint Configuration Thingy, Showing "Match Constraints" Sizing

Basic Anchoring, Dual Axis

Of course, `ConstraintLayout` can control sizing and positioning horizontally as well as vertically. The "URL dialog" scenario from the chapter on classic containers can be implemented easily enough using a `ConstraintLayout`:

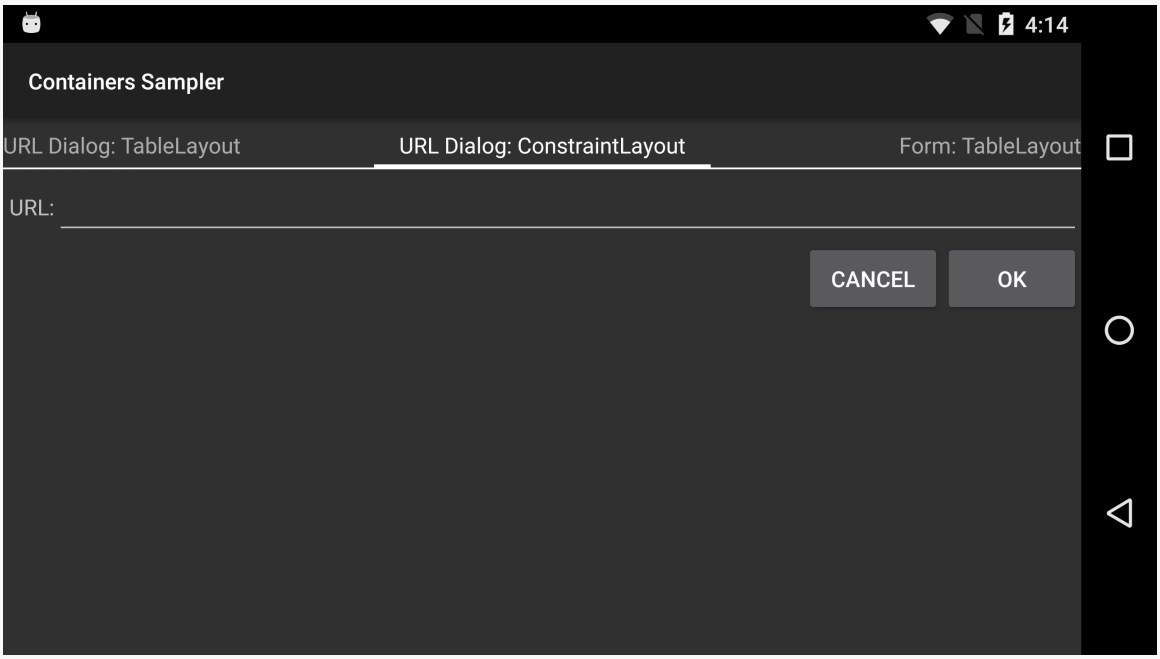


Figure 189: URL Dialog Layout, Using ConstraintLayout

Once again, the `ConstraintLayout` version resembles a more-complex representation of the `RelativeLayout` version:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="4dip"
    android:layout_marginStart="4dip"
    android:text="@string/url"
```

```
    app:layout_constraintBaseline_toBaselineOf="@+id/entry"
    app:layout_constraintStart_toStartOf="parent" />

  <EditText
    android:id="@id/entry"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:inputType="text"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@id/label"
    app:layout_constraintTop_toTopOf="parent" />

  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/ok"
    app:layout_constraintEnd_toEndOf="@id/entry"
    app:layout_constraintTop_toBottomOf="@id/entry" />

  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/cancel"
    app:layout_constraintEnd_toStartOf="@id/ok"
    app:layout_constraintTop_toTopOf="@id/ok" />

</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/url_dialog_cl.xml)

The `TextView` uses `app:layout_constraintStart_toStartOf="parent"` to say that its starting edge aligns with the starting edge of the `ConstraintLayout`. It also uses `app:layout_constraintBaseline_toBaselineOf="@+id/entry"` to have its baseline align with the baseline of the `EditText`.

The `EditText` uses:

- `app:layout_constraintStart_toEndOf="@id/label"` to have its starting edge align with the ending edge of the `TextView`
- `app:layout_constraintEnd_toEndOf="parent"` and `android:layout_width="0dp"` to have its ending edge align with the ending edge of the `ConstraintLayout` (and, therefore, span to fill the remaining space in the row)

- `app:layout_constraintTop_toTopOf="parent"` to have its top edge align with the top edge of the `ConstraintLayout`

The "OK" `Button` has its ending edge align with that of the `EditText` (`app:layout_constraintEnd_toEndOf="@id/entry"`) and has its top edge align with the bottom edge of the `EditText` (`app:layout_constraintTop_toBottomOf="@id/entry"`)

The "Cancel" `Button` has its ending edge align with the starting edge of the "OK" `Button` (`app:layout_constraintEnd_toStartOf="@id/ok"`) and has its top edge align with the top edge of the "OK" `Button` (`app:layout_constraintTop_toTopOf="@id/ok"`)

Note that to create a baseline via drag-and-drop, you need to:

- Click on the "ab" icon in the blueprint below the widget whose baseline should be constrained. This should enable a "lozenge" drag handle underneath the text in the widget:

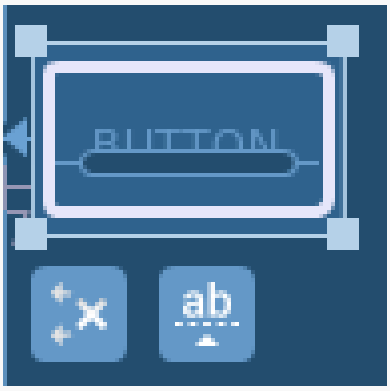


Figure 190: Baseline Constraint Enabled

- Start dragging that "lozenge" shape, which will enable similar shapes on other text-based widgets
- Drag the connection line to the "lozenge" in the other widget, to have the first widget's baseline depend upon the second widget's baseline:

Figure 191: Baseline Constraint Established

Guidelines

Where `ConstraintLayout` starts to depart from the classic containers is with its use of `Guideline` widgets.

Guidelines in graphics editors help in drawing operations. For example, in an SVG editor like Inkscape, you can "snap" shapes to align along a guideline. The guideline itself does not appear in the output; it is merely something used at drawing time to aid in drawing.

A `Guideline` serves a similar role with `ConstraintLayout`. A `Guideline` is not rendered as part of our UI. It does, however, allow us to anchor widgets inside the `ConstraintLayout` to the `Guideline`, just as we can anchor those widgets to other widgets or to the `ConstraintLayout` itself. And, as the name suggests, a `Guideline` is part of our GUI builder as well, to allow you to visually place the `Guideline` and connect widgets to that `Guideline`.

For example, in the chapter on the classic containers, we looked at how you could use weights with a `LinearLayout` to allocate size on a percentage basis. However, we did not replicate that structure with `RelativeLayout`, as `RelativeLayout` on its own has no way of working with percentages. `ConstraintLayout` on its own would also be incapable of handling this scenario... but with the use of a pair of `Guideline` widgets, it can:

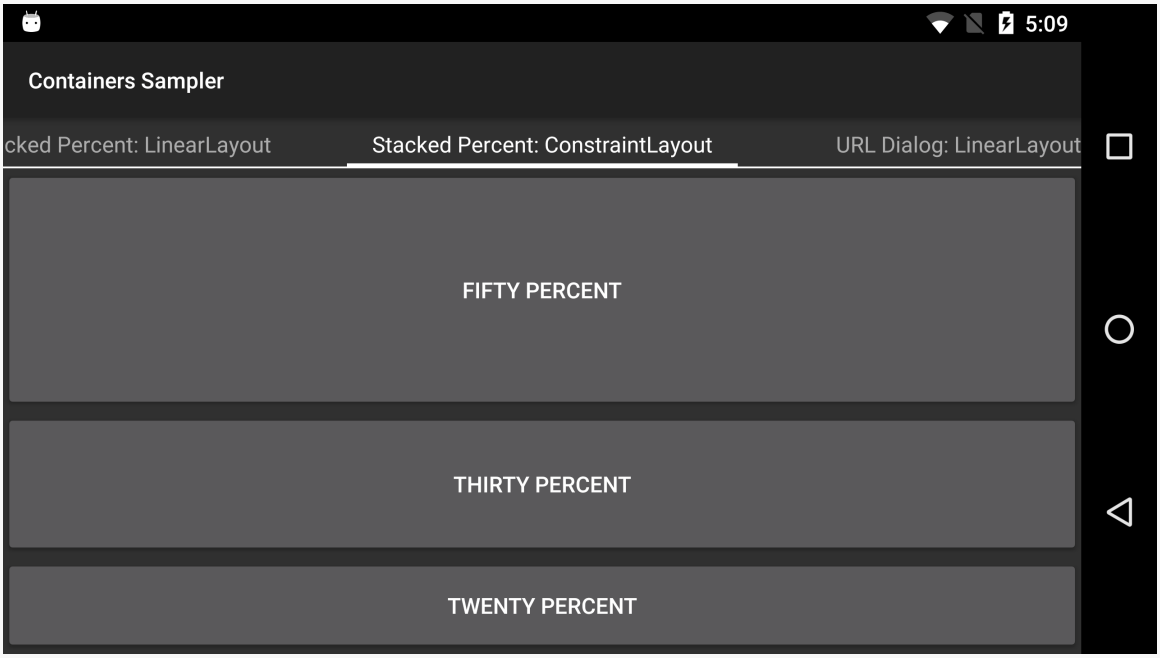


Figure 192: Stacked-Percent Layout, Using ConstraintLayout

And, as it turns out, `ConstraintLayout` does not require two passes through its children to attain this structure, the way that `LinearLayout` does when weights are applied.

The cost is that this layout is significantly more complex than is its `LinearLayout` counterpart:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.constraint.Guideline
    android:id="@+id/guideline_50"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    app:layout_constraintGuide_percent=".50" />

  <android.support.constraint.Guideline
    android:id="@+id/guideline_80"
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    app:layout_constraintGuide_percent=".80" />

  <Button
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:text="@string/fifty_percent"

    app:layout_constraintBottom_toTopOf="@id/guideline_50"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <Button
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:text="@string/thirty_percent"

    app:layout_constraintBottom_toTopOf="@id/guideline_80"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@id/guideline_50" />

  <Button
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:text="@string/twenty_percent"

    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@id/guideline_80" />

</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/stacked_percent_cl.xml)

A `Guideline` has an orientation, set via `android:orientation`. Both of the `Guideline` widgets used here are `horizontal`. This means that they span the width of the `ConstraintLayout`, and something else has to indicate where they reside vertically within the `ConstraintLayout`.

In this case, that "something else" is the `app:layout_constraintGuide_percent` attribute. This indicates how far down from the start of the `ConstraintLayout` these `horizontal` `Guideline` widgets belong, in terms of a fraction of the overall height of

the `ConstraintLayout`. In this case, they appear 50% (`.50`) and 80% (`.80`) from the start.

Our three `Button` widgets do not specify their own heights — they use `0dp` for `android:layout_height`. Nor do they express their heights in terms of weights, the way the `LinearLayout` equivalent does. Instead, they use the same anchoring rules that are used all the time... except now, they can anchor to those two `Guideline` widgets in addition to each other and the `ConstraintLayout`.

So, our top `Button` has its top edge aligned with the top edge of the `ConstraintLayout` (`app:layout_constraintTop_toTopOf="parent"`). But it also has its bottom anchored to the top of the `.50 Guideline` (`app:layout_constraintBottom_toTopOf="@id/guideline_50"`). Combined with the `0dp` height, this stretches the `Button` between the top and the middle of the `ConstraintLayout`.

The middle `Button` anchors itself between the two `Guideline` widgets, using `app:layout_constraintBottom_toTopOf="@id/guideline_80"` and `app:layout_constraintTop_toTopOf="@id/guideline_50"`. It too stretches to fill that gap, and so it will take up 30% of the space (between the `.50` and the `.80` `Guideline` widgets).

Similarly, the bottom `Button` ties itself to the bottom of the `ConstraintLayout` (`app:layout_constraintBottom_toBottomOf="parent"`) as well as to the `.80` `Guideline` (`app:layout_constraintTop_toTopOf="@id/guideline_80"`).

To add a guideline through the graphical layout editor, use the toolbar button that looks like a vertical dotted line between two small solid horizontal lines. Clicking that opens up a drop-down to choose a vertical or a horizontal guideline:

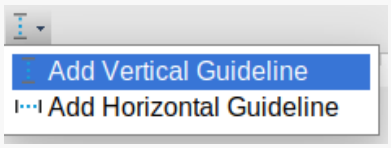


Figure 193: Guideline Toolbar Button, With Drop-Down

Choosing one of the guideline options drops a guideline in the desired orientation into the blueprint view, which you can then drag to the desired location.

Converting Existing Layouts

After spending some time with `ConstraintLayout`, you might decide that you want to try to standardize on it, converting existing layouts based on other containers to use `ConstraintLayout`.

Android Studio can help with this. In the design tab of a layout, if you right-click over an existing container in the component tree, you will have a "Convert ... to ConstraintLayout" context menu option. The "..." will be replaced by the container class that you clicked on (e.g., `LinearLayout`).

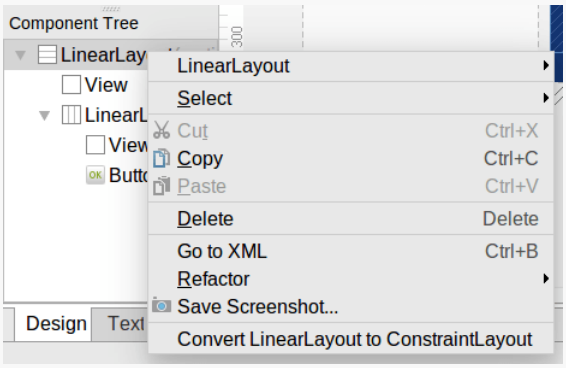


Figure 194: Convert to ConstraintLayout Context Menu

Choosing that menu option brings up a dialog for configuring the conversion:

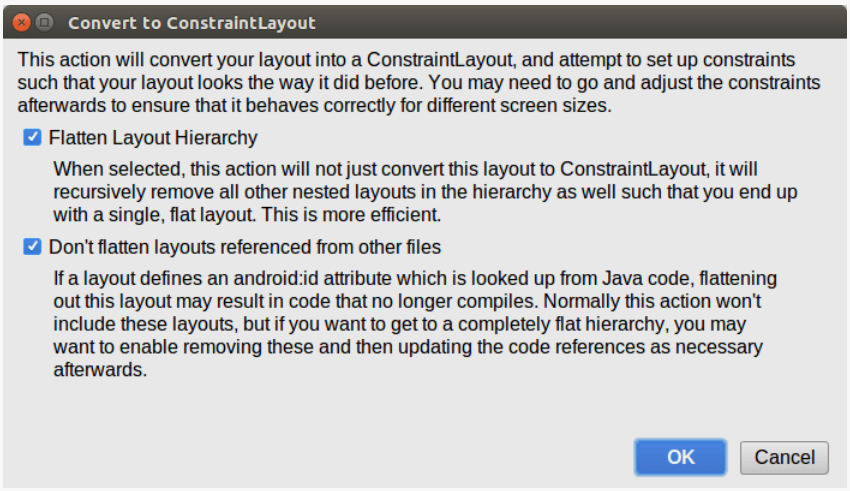


Figure 195: Convert to ConstraintLayout Dialog

The default behavior is to not only change the container that you clicked on, but all containers inside of it, to try to put everything into a single `ConstraintLayout`. However, if you have Java code that refers to some container that this conversion process would otherwise remove — for example, you are calling `findViewById()` on a widget ID that your Java code references — that container is left alone. Unchecking the second checkbox results in a more aggressive conversion, but it will leave you with broken Java code until you change whatever logic you have there to deal with the revised layout.

For simple layouts, the conversion process works fairly well. For complex layouts, the conversion process is more likely to give you invalid results, requiring manual tinkering, reversion to the original layout, or rewriting the layout from scratch.

For example, converting the `LinearLayout` edition of the complex form results in a layout that does not resemble the original:

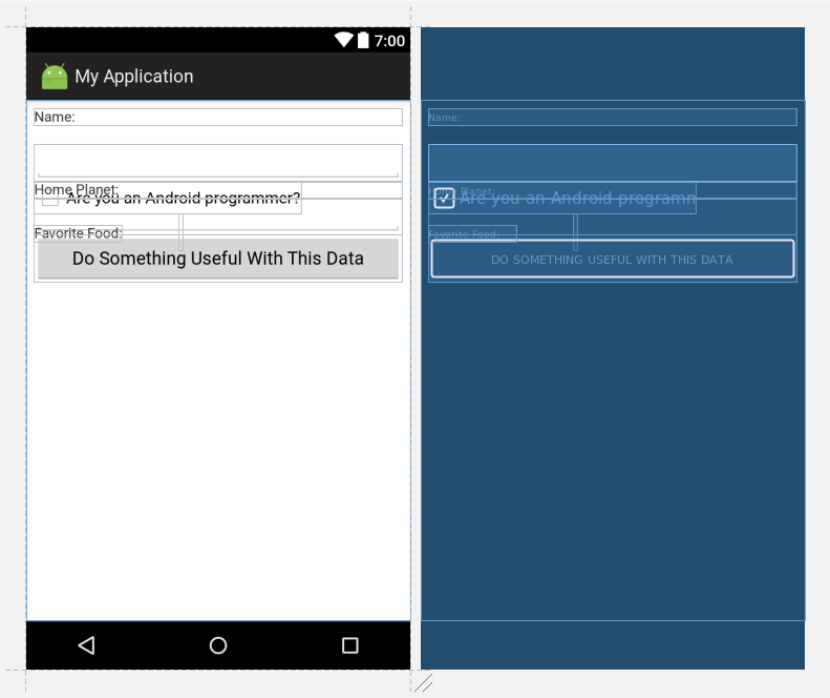


Figure 196: Complex LinearLayout Form, After Conversion

Attempting to convert the `TableLayout` edition of that same form results in a layout that will not compile properly.

Even seemingly simple layouts, such as the `RelativeLayout` overlap layout, fail to convert correctly:

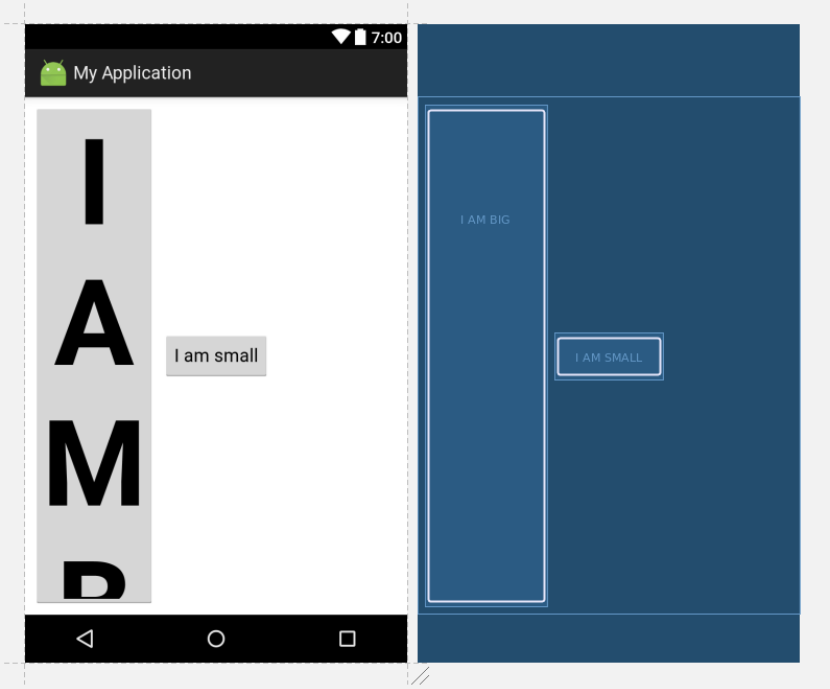


Figure 197: Not-So-Complex RelativeLayout Form, After Conversion

So, you are welcome to try it out and see if it works, but do not be surprised if it does not.

Visit the Trails!

There are more things that you can accomplish with a `ConstraintLayout` beyond what is presented here. There is a chapter on advanced `ConstraintLayout` techniques that get into more complex scenarios.

If you are interested in other containers from libraries, the book has a chapter on `GridLayout`.

The Action Bar

The action bar — that bar that runs across the top of your activity — is the backbone of your UI. Here, you can provide actions for the user to perform related to the current activity (e.g., "edit the contact that you are viewing") or related to the application as a whole (e.g., "here is the documentation"). Sometimes, these actions will appear as toolbar buttons or other widgets in the action bar. Sometimes, these actions will appear in the "overflow", which amounts to a menu.

This chapter introduces the concept of the action bar and how to add actions to it.

Bar Hopping

Android has had many patterns for various "bars" as part of its UI. So, to help explain what an action bar is, it helps if we review the history and role of Android's various bars.

Android 1.x/2.x

In the beginning, there was the status bar and the title bar.

The status bar was a thin strip across the top of the screen, used for things like the clock, signal strength, battery charge, and notification icons (for events like new unread email messages). This bar is technically part of the OS, not your app's UI.

The title bar was a thin gray strip beneath the status bar that, by default, would hold the name of your application, much like the title bar of a browser might show the name of a Web site.

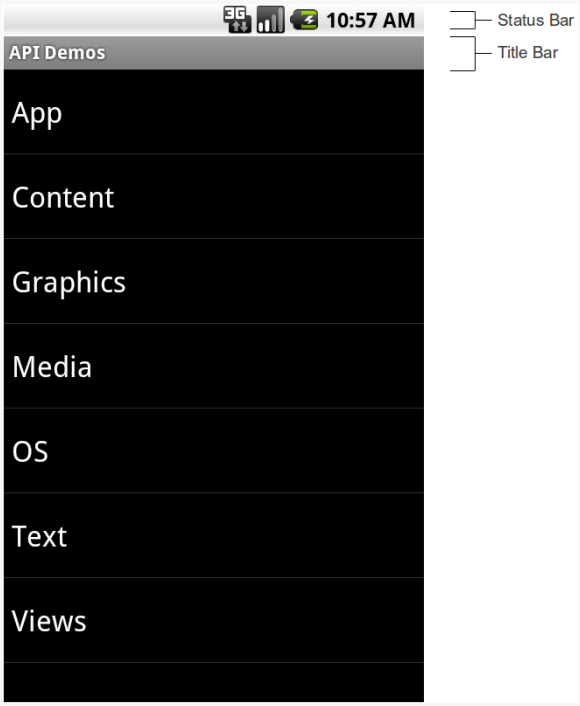


Figure 198: Status Bar and Title Bar

Android 3.0-4.1, Tablets

When official support for tablets arrived with Android 3.0 in February 2011, the story changed.

The status bar was replaced by the system bar, appearing at the bottom of the screen. This had all of the contents of the old status bar, but also had the soft keys for BACK, HOME, etc. Android 1.x and 2.x required that devices have off-screen affordances for those operations; now, device manufacturers could skip those and have the system bar offer them.

The action bar, by default, appears at the top of your activity, replacing the old title bar. You can define what goes in the action bar (icon, title, toolbar buttons, etc.).

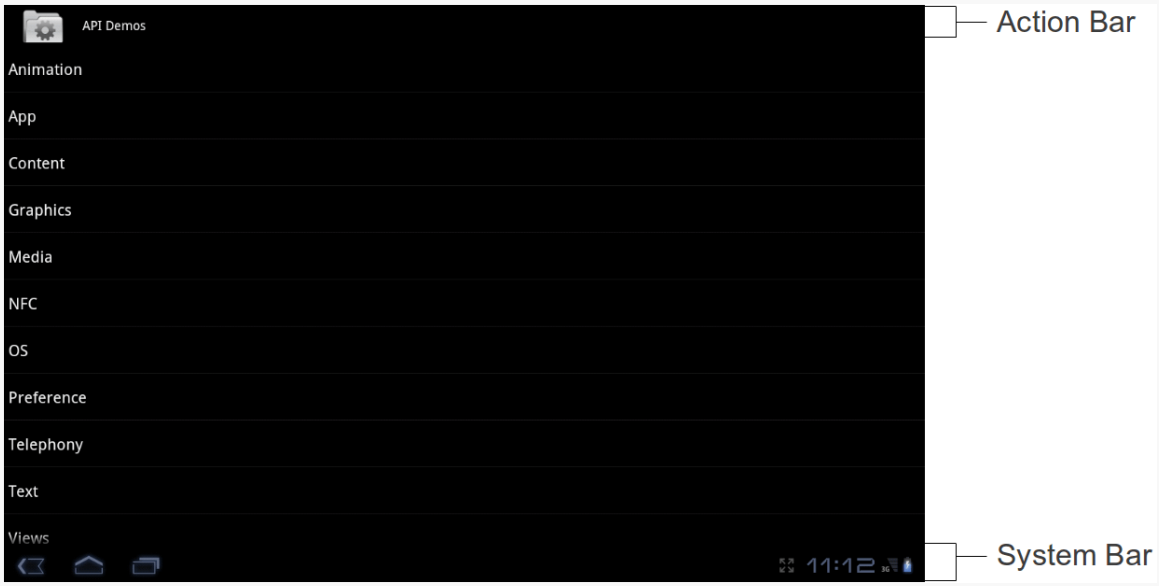


Figure 199: Action Bar and System Bar

The icon on the far left of the action bar also serves as a toolbar button, if you wish. A common pattern for using this is to take the user back to the "main" or "home" activity of your application.

Sometimes, the far right side of the action bar will contain a "..." affordance. This is known as the "action overflow" or "overflow menu":

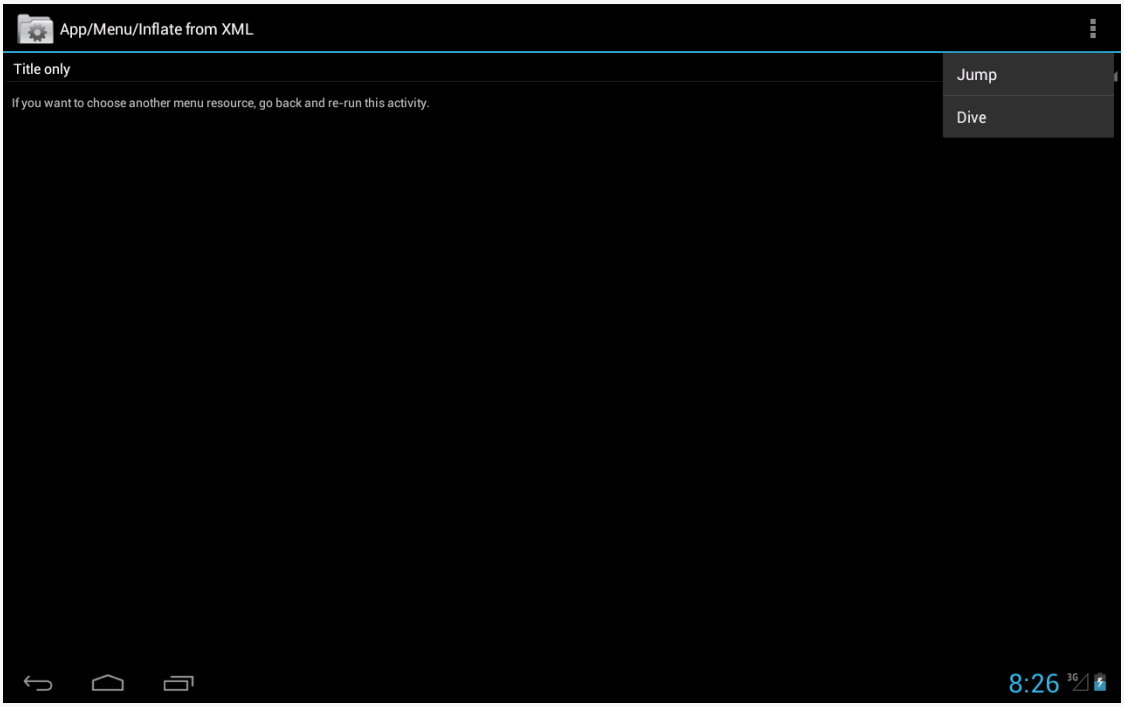


Figure 200: Action Bar with Open Overflow Menu

Tapping it will give the user access to actions that might have been toolbar buttons on a larger screen, but there was insufficient room. Also, low-priority actions may be tucked into the overflow, rather than clutter up the screen with too many toolbar buttons.

Android 4.0-4.4, Phones

Phone-sized devices were not supported by Android 3.x. They jumped from Android 2.3 to 4.0, and along the way adopted some of the Android 3.x UI features:

- Phone apps could have an action bar, like their tablet counterparts
- Device manufacturers could skip the BACK, HOME, etc. buttons and let a partial system bar handle those
- The status bar remained intact from the Android 2.x approach

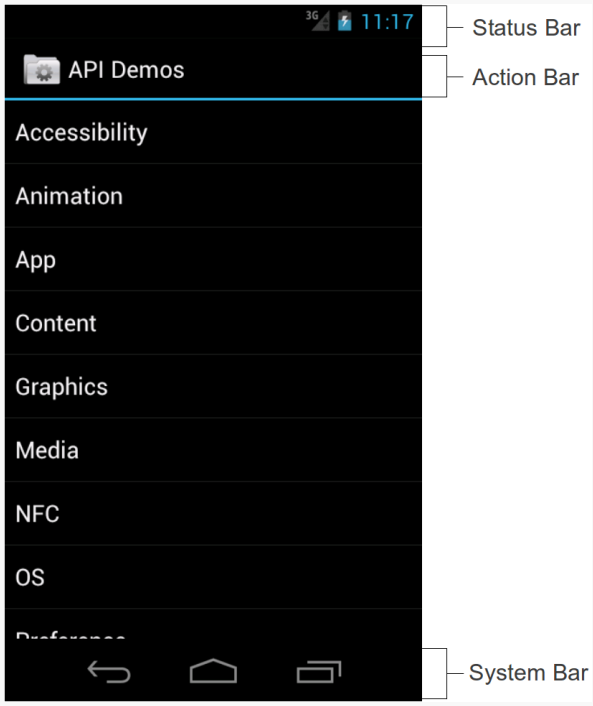


Figure 201: Status Bar, Action Bar, and System Bar

Android 4.2-4.4, Tablets

The Nexus 7, introduced in the summer of 2012, was a 7" tablet that did not follow the tablet UI structure that all other standard Android tablets used. Instead, it looked a bit like a really large phone, having a top status bar along with a bottom system bar solely for the navigation buttons (BACK, HOME, etc.). Apps, as before, could have an action bar as well.

Initially, it was thought that the Nexus 7 was going to be distinctive in that regard. Instead, with Android 4.2, Google switched all tablets to this model, restoring the status bar and relegating the system bar purely for navigation buttons.

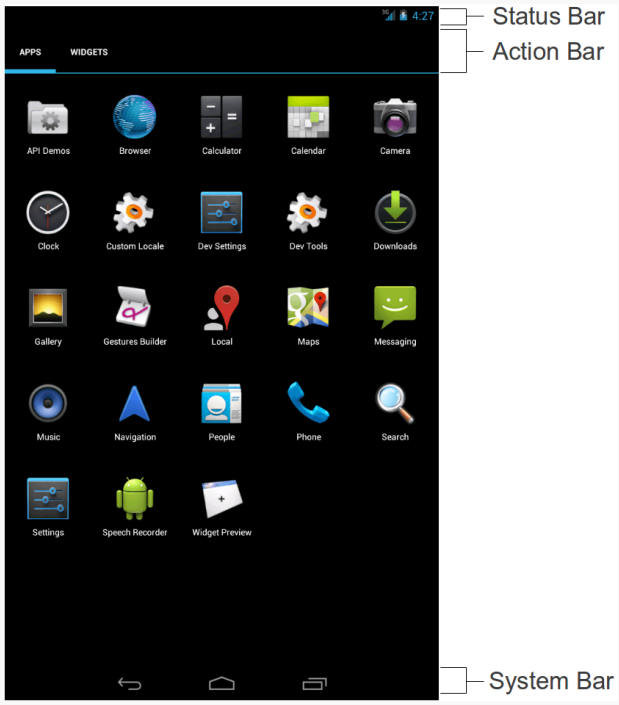


Figure 202: Status Bar, Action Bar, and System Bar, on Nexus 7 Emulator

Android 5.0+

Functionally, the action bar is much the same in Android 5.0 as it was in previous releases. However, aesthetically, it has dropped the icon and made other minor stylistic adjustments.

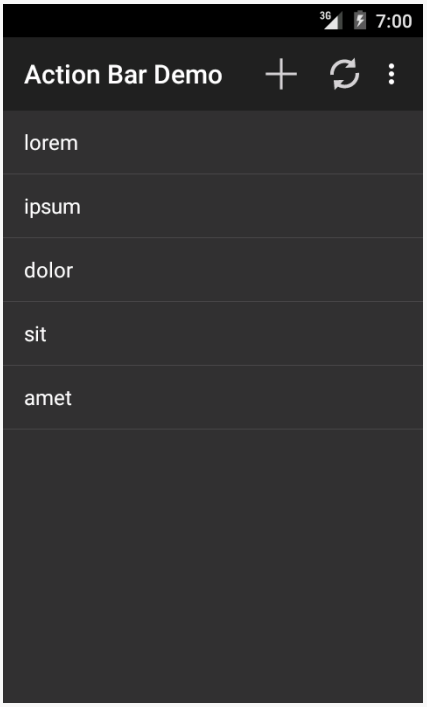


Figure 203: Action Bar on Android 5.0 Emulator

Yet Another History Lesson

Back in the dawn of Android time, referred to by some as "the year 2007", we had options menus. These would rise up from the bottom of the screen based on the user pressing a MENU key:

Figure 204: Legacy Options Menu

This is why you will see references to "options menu" scattered throughout the Android SDK.

The action bar pattern was first espoused by Google at the 2010 Google I|O conference. However, at the time, there was no actual implementation of this, except in scattered apps, and definitely not in the Android SDK.

Android 3.0 — a.k.a., API Level 11 — added the action bar to the SDK, and apps targeting that API level will get an action bar when running on such devices.

Your Action Bar Options

There are several implementations of the action bar floating about. You will probably be using the one that is part of Android itself, starting with API Level 11. However, there are a couple of backports of the action bar if you need them.

Pure Native

As mentioned above, devices running Android 3.0 and higher have support for the action bar as part of their firmware, and that support is exposed through the Android SDK. For example, there is an `ActionBar` class, and you can get an instance of it for your activity's action bar via `getActionBar()`.

However, this only works on devices running Android 3.0 and higher. If you try calling `getActionBar()` on an older device, you will crash with a `VerifyError` runtime exception. `VerifyError` is Android's way of telling you "while you compiled fine, something your compiled code refers to does not exist".

If your `minSdkVersion` is 11 or higher, you will be able to use the native action bar, and that approach will be used in most of this book.

Backports

If your `minSdkVersion` is lower than 11, you have two major choices:

1. Use the "menu" APIs in Android, which will add stuff to the action bar on newer devices, but will result in the classic "options menu" on older devices.
2. Use the `appcompat-v7` backport of the action bar, published by Google in the Android Support package in August 2013.

This chapter assumes that your `minSdkVersion` is set to 11 or higher and you will use the native action bar. A separate chapter in the trails cover the use of `appcompat-v7`.

Note that the `appcompat-v7` library not only backports the action bar, but also attempts to backport part of Google's Material Design styling. Normally, Material Design only comes from Android 5.0 and the use of `Theme.Material`. The `appcompat-v7` chapter will cover the library's effects both to the action bar and to other aspects of your app's UI.

A Quick Note About Toasts

In the sample app that follows, we use a `Toast` to let the user know some work has been completed.

A `Toast` is a transient message, meaning that it displays and disappears on its own without user interaction. Moreover, it does not take focus away from the currently-

active `Activity`, so if the user is busy writing the next Great Programming Guide, they will not have keystrokes be "eaten" by the message.

Since a `Toast` is transient, you have no way of knowing if the user even notices it. You get no acknowledgment from them, nor does the message stick around for a long time to pester the user. Hence, the `Toast` is mostly for advisory messages, such as indicating a long-running background task is completed, the battery has dropped to a low-but-not-too-low level, etc.

Making a `Toast` is fairly easy. The `Toast` class offers a static `makeText()` method that accepts a String (or string resource ID) and returns a `Toast` instance. The `makeText()` method also needs the `Activity` (or other `Context`) plus a duration. The duration is expressed in the form of the `LENGTH_SHORT` or `LENGTH_LONG` constants to indicate, on a relative basis, how long the message should remain visible. Once your `Toast` is configured, call its `show()` method, and the message will be displayed.

Setting the Target

If you want proper action bar support, you will want to target API Level 14 or higher at runtime. That involves setting the `targetSdkVersion` property in your `build.gradle` file (for Android Studio users) or setting the `android:targetSdkVersion` attribute of the `<uses-sdk>` element of your manifest (for legacy pre-Gradle projects).

We see this in the manifest of the [ActionBar/ActionBarDemoNative] sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.inflation"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="10"
    android:targetSdkVersion="19"/>

  <application
```

```
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from ActionBar/ActionBarDemoNative/app/src/main/AndroidManifest.xml)

Specifically, we have `android:targetSdkVersion` set to 19. While 11 or higher will give you an action bar, 14 or higher will solve a particular UI quirk related to menu choices. Some Android 4.0+ devices, but not all, will show *two* ways of getting at overflow menu items if you have your `android:targetSdkVersion` set to a value between 11 and 13. You will have the "..." item in the action bar itself *and* a second one in the system bar, on devices that have one. Setting `android:targetSdkVersion` to 14 or higher resolves this.

Doing nothing else but the preceding steps would give us an action bar, but one with no toolbar icons or action overflow menu. While perhaps visually appealing, this is not terribly useful for the user, so we need to do some more work to give the user actions to perform from the action bar.

Note that this manifest has a `minSdkVersion` of 10. This means that the app can run on Android 2.3.3 devices. On those devices, though, the app will not have an action bar, as the action bar did not exist then, and this app is not using a backport like `appcompat-v7`. Instead, the app will have an old-style options menu on API Level 10 devices. There is nothing intrinsically wrong with this, though it does mean that your app will look different on API Level 10 devices.

Defining the Resource

The easiest way to get toolbar icons and action overflow items into the action bar is by way of a menu XML resource. This is called a "menu" resource for historical reasons, as these resources originally were used for things like the options menu.

You can add a `res/menu/` directory to your project and place in there menu XML resources, such as `res/menu/actions.xml` from `ActionBar/ActionBarDemoNative`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/add"
    android:icon="@drawable/ic_action_new"
    android:showAsAction="always"
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@drawable/ic_action_refresh"
    android:showAsAction="always|withText"
    android:title="@string/reset"/>
  <item
    android:id="@+id/about"
    android:icon="@drawable/ic_action_about"
    android:showAsAction="never"
    android:title="@string/about">
  </item>

</menu>
```

(from ActionBar/ActionBarDemoNative/app/src/main/res/menu/actions.xml)

There are four things you will want to configure on every menu item (`<item>` element in the XML):

1. The ID of the item (via the `android:id` attribute in XML). This will create another `R.id` value, associated with this menu item, much like the `R.id` values for our widgets in our layouts. We will use this ID to determine when the user clicks on one of our toolbar buttons or action overflow items.
2. The title of the item (via the `android:title` attribute in XML). If this item winds up in the action overflow menu, or optionally as part of its toolbar button, this text will appear. Also, this title will appear as a "tooltip" on the action item in the action bar itself, if the user long-presses on the icon (something few users know to do). Typically, you will use a string resource reference (e.g., `@string/add`), to better support internationalization.
3. The icon for the item (via the `android:icon` attribute in XML). If your item will appear as a toolbar button, this icon is used with that button.
4. Flags indicating how this item should be portrayed in the action bar (via the `android:showAsAction` attribute in XML). You will choose to have it be

`always` a toolbar button, only be a toolbar button `ifRoom`, or have it `never` be a toolbar button. You can also elect to append `|withText` to either `always` or `ifRoom`, to indicate that you want the toolbar button to be both the icon and the title, not just the icon. Note that `always` is not *guaranteed* to be a toolbar button — if you ask for 100 `always` items, you will not have room for all of them. However, `always` items get priority for space in the action bar over `ifRoom` items.

A Quick Note About Android Studio

Android Studio 2.2 introduced a new menu editor, modeled after the graphical layout editor. When you open a menu resource, you get two sub-tabs: a Text one with the XML, and a Design one to preview the menu and, in theory, edit it:

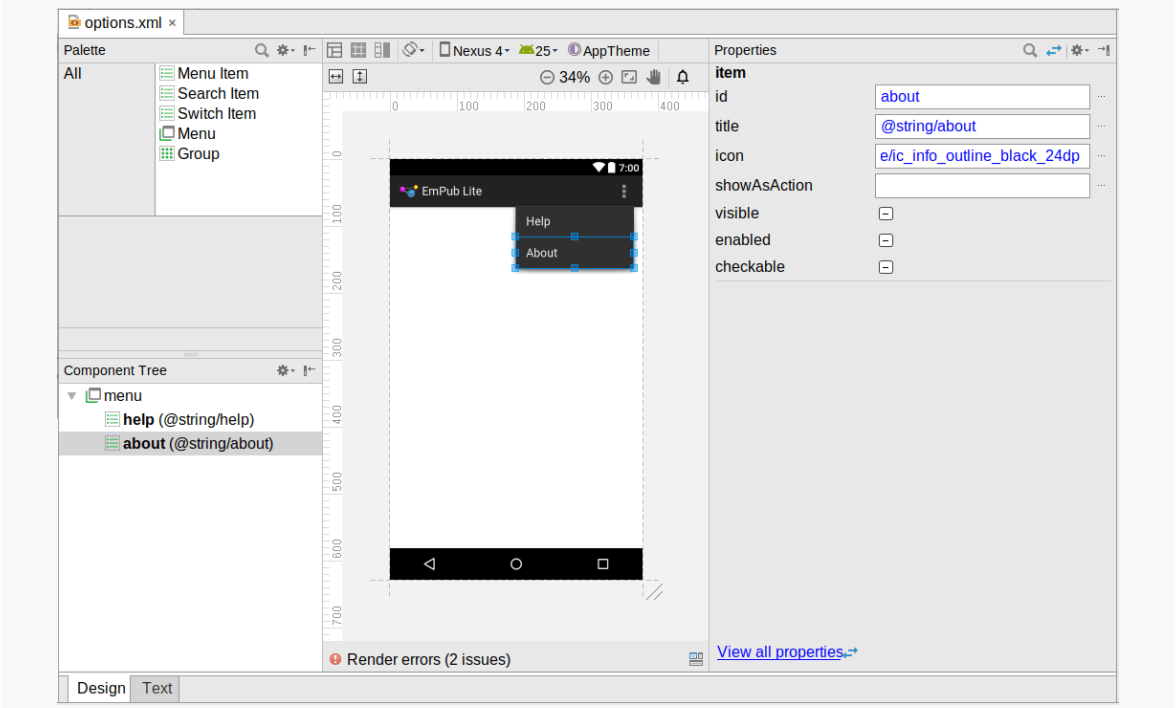


Figure 205: Android Studio Graphical Menu Editor

Unfortunately, the Design sub-tab suffers from this bug, making it annoying to use. For the time being, you are better off working with the XML directly.

Applying the Resource

From your activity, you teach Android about these action bar items by overriding an `onCreateOptionsMenu()` method, such as this one from the `ActionBarDemoActivity` of the `ActionBar/ActionBarDemoNative` sample project:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  return(super.onCreateOptionsMenu(menu));
}
```

(from ActionBar/ActionBarDemoNative/app/src/main/java/com/commonsware/android/inflation/ActionBarDemoActivity.java)

Here, we create a `MenuInflater` and tell it to inflate our menu XML resource (`R.menu.actions`) and pour them into the supplied `Menu` object. We then chain to the superclass, returning its result.

Responding to Events

To find out when the user taps on one of these things, you will need to override `onOptionsItemSelected()`, such as the `ActionBarDemoActivity` implementation shown below:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.add:
      addWord();

      return(true);

    case R.id.reset:
      initAdapter();

      return(true);

    case R.id.about:
      Toast.makeText(this, R.string.about_toast, Toast.LENGTH_LONG)
           .show();

      return(true);
  }
```

```
    return(super.onOptionsItemSelected(item));
  }
```

(from ActionBar/ActionBarDemoNative/app/src/main/java/com/commonsware/android/inflation/ActionBarDemoActivity.java)

You will be passed a `MenuItem`. You can call `getItemId()` on it and compare that value to the ones from your menu XML resource (`R.id.add` and `R.id.reset`). If you handle the event, return `true`; otherwise, return the value of chaining to the superclass' implementation of the method.

If you wish to respond to taps on your application icon, on the left of the action bar, compare `getItemId()` to `android.R.id.home`, as that will be the `MenuItem` used for that particular toolbar button. Note that if you have your `android:targetSdkVersion` set to `14` or higher, you will also need to call `setHomeButtonEnabled(true)` on the `ActionBar` (obtained via a call to `getActionBar()`) to enable this behavior. Note that this icon may not exist, particularly if you are using `Theme.Material` on Android 5.0+.

The Rest of the Sample Activity

So, what is it that we really are doing here in `ActionBarDemoActivity`?

In many respects, this is reminiscent of the `ListActivity` demos from an earlier chapter. We have an array of 25 Latin words, and we want to display these in a list.

However, in this case, we are only showing five words at the outset. An "add" action bar item will add additional words out of the main roster of 25 words, until the `ListView` holds all 25. A "reset" action bar item will return us to the original 5 words.

`ActionBarDemoActivity` is a `ListActivity`. However, rather than set up our `ArrayAdapter` directly in the `onCreate()` method as some of the other samples have done, we delegate that work to an `initAdapter()` method. Moreover, that `initAdapter()` method does its work a bit differently than what those other samples did:

```
  private void initAdapter() {
    words=new ArrayList<String>();

    for (int i=0;i<5;i++) {
      words.add(items[i]);
    }
```

```
    adapter=
        new ArrayAdapter<String>(this,
                                 android.R.layout.simple_list_item_1,
                                 words);

    setListAdapter(adapter);
  }
```

(from ActionBar/ActionBarDemoNative/app/src/main/java/com/commonsware/android/inflation/ActionBarDemoActivity.java)

Rather than create the `ArrayAdapter` straight out of the static `items` array, we create a fresh `ArrayList` and pour the 5 elements from `items` into it, then create the `ArrayAdapter` on the `ArrayList`. This may seem superfluous, but we will take advantage of this approach with our action bar items.

When the user clicks the "reset" item in the action bar, we call `initAdapter()` again, which gives our `ListActivity` a fresh set of 5 Latin words to display:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.add:
      addWord();

      return(true);

    case R.id.reset:
      initAdapter();

      return(true);

    case R.id.about:
      Toast.makeText(this, R.string.about_toast, Toast.LENGTH_LONG)
           .show();

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from ActionBar/ActionBarDemoNative/app/src/main/java/com/commonsware/android/inflation/ActionBarDemoActivity.java)

When the user clicks the "add" item in the action bar, we call an `addWord()` private method, which adds the next word out of the `items` array and appends it to the `ListView`:

```
private void addWord() {
  if (adapter.getCount()<items.length) {
    adapter.add(items[adapter.getCount()]);
  }
}
```

(from ActionBar/ActionBarDemoNative/app/src/main/java/com/commonsware/android/inflation/ActionBarDemoActivity.java)

The net result of all of this is that we have an activity with our customized action bar:

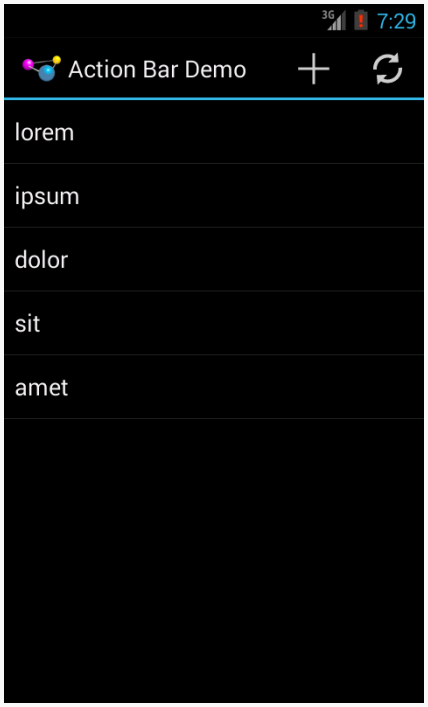


Figure 206: ActionBarDemo, As Initially Launched, on Android 4.3

Among our action bar items is an "about" one that will always be in the overflow menu. This will have three possible visual outcomes.

First, on devices without an off-screen MENU key, the overflow menu is represented by a "..." button, which displays the overflow menu when clicked:

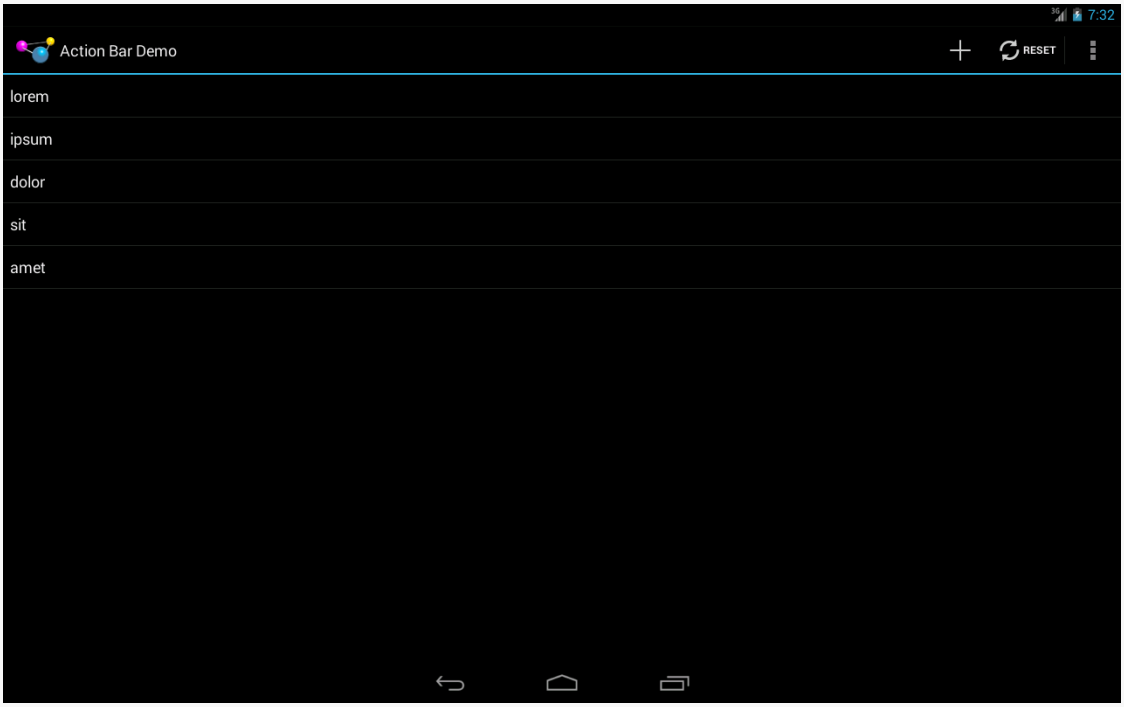


Figure 207: ActionBarDemo, on Android 4.3 Large Screen, with Overflow

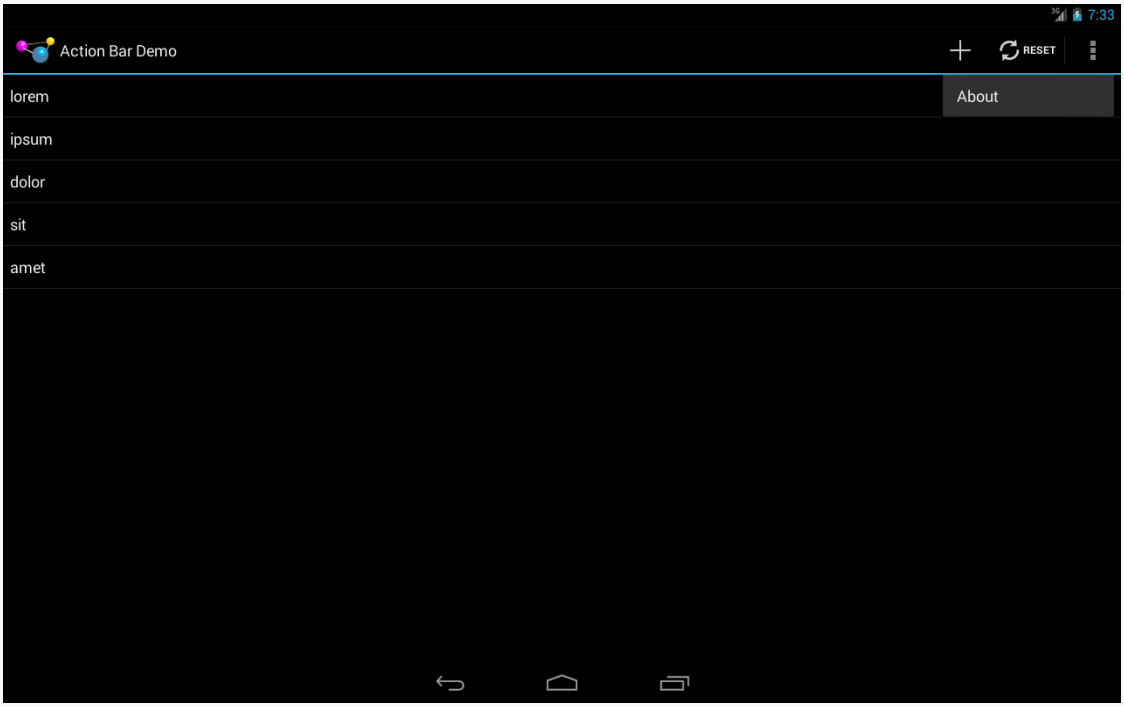


Figure 208: ActionBarDemo, on Android 4.3 Large Screen, with Overflow Open

On Android 4.x devices with an off-screen MENU key, pressing the MENU key will cause the overflow menu to rise up from the bottom of the screen:

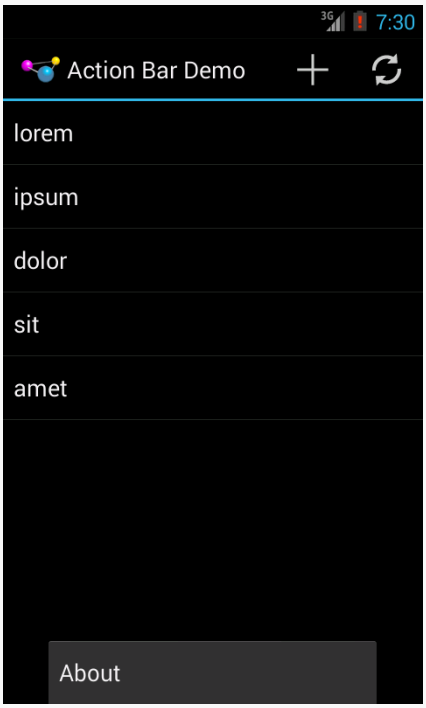


Figure 209: ActionBarDemo, on Android 4.3 Normal Screen, with Overflow

Android 4.4+ devices should always have the "..." button, as is described in the next section.

Android 2.3 devices that run this app will have no action bar:

Figure 210: ActionBarDemo, on Android 2.3.3 Normal Screen

However, pressing the MENU button will bring up the old-style options menu, where our action items appear:

Figure 211: ActionBarDemo, on Android 2.3.3 Normal Screen, Showing the Options Menu

MENU Key, We Hardly Knew Ye

To expand upon the history lessons from earlier in this chapter, all Android 1.x and 2.x devices had a MENU key, used to bring up the options menu. With Android 3.0 and the advent of the system/navigation bar, device manufacturers no longer needed keys for HOME, BACK, and MENU. And, the action bar incorporated a "..." affordance for accessing the overflow, for items that would have been in the options menu and were not promoted to be toolbar buttons in the action bar itself.

Confusion began when we started having devices that *had* a MENU key *and* Android 3.0+. A few Android 2.x devices were upgraded to Android 4.0, and hundreds of millions of Android devices, from manufacturers like Samsung and HTC, shipped with Android 4.x and a MENU key.

To accommodate this, the device would report whether it had a "permanent menu key", and the action bar would choose whether to show the "..." affordance based upon the existence of this key. Devices with a MENU key would not get the "...", but instead would use the MENU key to display the overflow.

This irritated many developers, for much the same reason as why the MENU key irritated those developers back in Android 1.x/2.x: the existence of a menu was not very discoverable. Many users would eventually realize that tapping the MENU key might uncover useful stuff, but not all users would make this connection. However, now developers could see an obvious alternative, in the form of the "..." affordance, and so they sought ways to trick the action bar into showing the "..." even on devices that had a MENU key.

And that was how the world worked... up until Android 4.4.

An unannounced change in Android 4.4 is that the "..." button should now *always* be shown in the action bar. The MENU key, if it exists, will still work, showing the overflow. Ideally, it shows the overflow as dropping down from the "...", though that is not required. And the Compatibility Definition Document for Android 4.4 more forcefully suggests that the MENU key is obsolete.

None of this should directly affect your code. However:

- When taking screenshots, bear in mind that they will vary between devices that have the "..." button and those that do not
- When writing documentation, or blog posts, or other instructional material, try to phrase references to the overflow that will work for both those users with a "..." button and those that do not

Action Bars, Live in Living Color!

On Android 4.0+, if you are using a `Holo` theme as a base, you may wish to adjust the colors used by your action bar.

On Android 5.0+, if you are using a `Material` theme as a base, you *will* want to adjust the colors used by your action bar. This is Google's vision for how branding should work, in lieu of having your icon be in the action bar.

The following sections outline some ways to affect the colors of your action bar.

Material Tint Effects

Android 5.0 and `Theme.Material` make action bar colors easy to set up, as part of an overall "tinting" approach.

The ActionBar/MaterialColor sample project is a clone of the ActionBarDemoNative sample shown earlier in this chapter, but one where:

- Our minSdkVersion is set to 21, so the app will only run on Android 5.0+
- We set up a custom theme, with specific tinting rules, that affect our action bar colors

Color Resources

The theme will need to refer to colors, and the cleanest way to do that is to set up color resources. Like all of our other resources, we give color resources a name and a color value, usually in #RRGGBB or #AARRGGBB format. Color resources are "value" resources, held by default in res/values/, with the convention of using a colors.xml file for the actual colors.

For example, here is the res/values/colors.xml file from the MaterialColor sample application:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#3f51b5</color>
  <color name="primary_dark">#1a237e</color>
  <color name="accent">#ffee58</color>
</resources>
```

(from ActionBar/MaterialColor/app/src/main/res/values/colors.xml)

It defines three colors, primary, primary_dark, and accent, with different colors for each. In Android Studio, editing this file shows a tiny color swatch to help you visualize the colors:

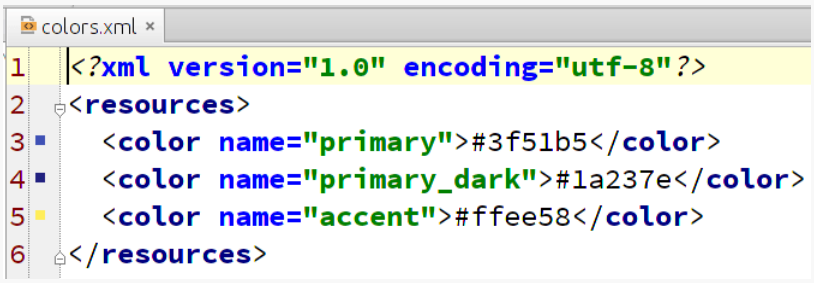


Figure 212: Color Resources in Android Studio

Tinting a Theme

Then, given that we have definitions of our colors, we can apply those colors to a custom theme, found in `res/values/styles.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

(from ActionBar/MaterialColor/app/src/main/res/values/styles.xml)

Here, our `AppTheme` is inheriting from `Theme.Material` and is overriding three tints: `colorPrimary`, `colorPrimaryDark`, and `colorAccent`, referring to our three color resources in turn.

Note that we could have inherited from `Theme.Material.Light` had we wanted a light "content area" (where our widgets go), or even `Theme.Material.Light.DarkActionBar` for a light content area and a dark action bar (before we start tailoring the action bar colors).

Applying the Theme

The application's manifest declares that we will use `AppTheme` as the default theme for our `<application>`, so all activities will use that theme unless overridden at the activity level:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.abmatcolor"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
```

```
    android:minSdkVersion="21"
    android:targetSdkVersion="21" />

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from ActionBar/MaterialColor/app/src/main/AndroidManifest.xml)

Also note that here is where we specify that `minSdkVersion` is 21. A new Android Studio project would do that in `build.gradle`.

The Results

Everything else about the app is the same as the `ActionBarDemoNative` sample, including our activity and the `ListView` that we are populating.

However, when we run this edition on an Android 5.0+ device or emulator, our action bar takes on the requested colors, specifically the `colorPrimary` value for the background color of the action bar:

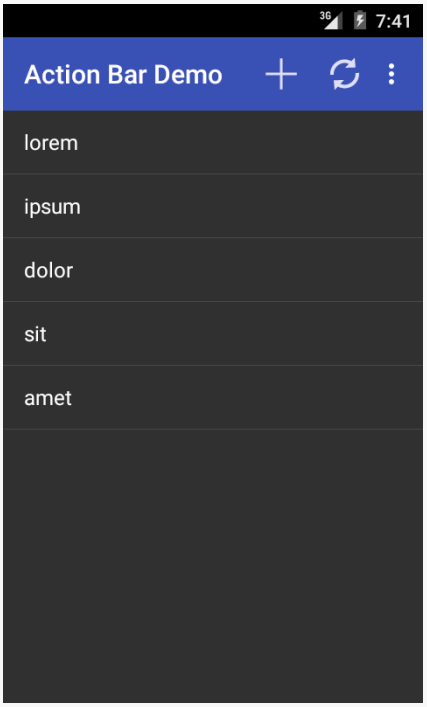


Figure 213: MaterialColor on Android 5.0 Emulator

The custom theme also affects the colors of certain widgets, as will be covered later in the book.

Restoring the Icon (Sort Of)

While the Material Design philosophy skips the application icon that we used to have in the action bar, there is a way to add it back for a `Theme.Material` application, though it requires a little bit of work, as seen in the `ActionBar/MaterialLogo` sample project.

The key thing that you need to do is to call `setDisplayShowHomeEnabled(true)` on your `ActionBar` object, which you get by calling `getActionBar()` in your `Activity`:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  getActionBar().setDisplayShowHomeEnabled(true);

  initAdapter();
}
```

(from ActionBar/MaterialLogo/app/src/main/java/com/commonsware/android/abmatlogo/ActionBarDemoActivity.java)

This will use whatever icon is set for the `android:icon` attribute in your manifest as the "home" icon in your action bar:

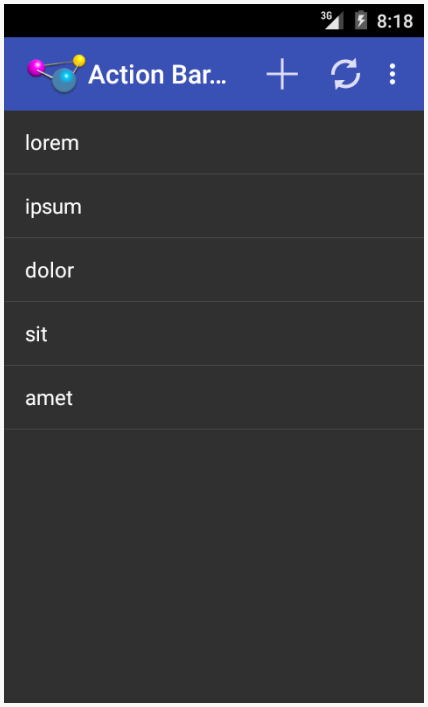


Figure 214: MaterialLogo on Android 5.0 Emulator

If you would rather use a different icon, such as one that is scaled to fit the action bar a bit better, you can call `setIcon()` on your `ActionBar`, supplying the ID of a drawable resource (e.g., `R.drawable.action_bar_icon`) that should be used instead of the drawable specified in the `android:icon` attribute of your `<activity>` or `<application>` in the manifest.

Action Bar Style Generator

For `Theme.Holo` and kin, the tinting rules from `Theme.Material` will not apply. Instead, you will need to do a fair bit of tinkering to get the color scheme set up the way you want.

Or, you can use Jeff Gilfelt's Action Bar Style Generator.

This is a Web site that allows you to design an action bar color scheme, where the site will then generate for you everything that you need to implement that color scheme.

Note that Mr. Gilfelt has marked this site as deprecated, with an eye towards people using `Theme.Material` or the `appcompat-v7` edition of the action bar. The site works, but in all likelihood it will be discontinued at some future date.

Also note that the site works best with Google's Chrome or Chromium browsers, though in testing, a recent edition of Firefox worked as well. As the site indicates, "your mileage may vary with other browsers".

Designing the Scheme

The site is dominated by a form for designing the color scheme and a preview area to show what the design will look like:

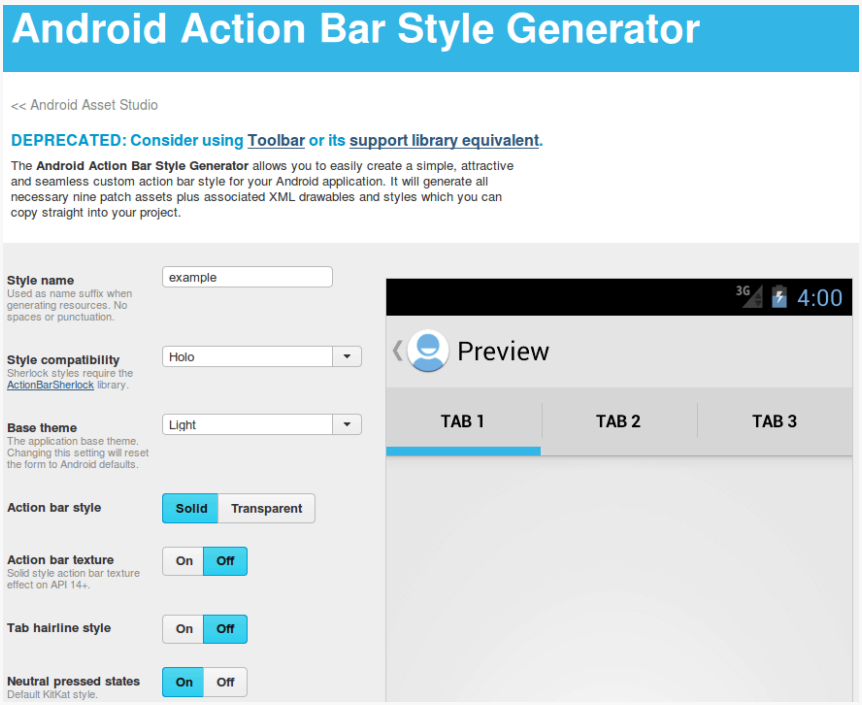


Figure 215: Action Bar Style Generator, As Originally Launched

In the "Style name" field, you can fill in the name you want to give your custom theme. Whatever you fill in will be converted into all lowercase with a leading

capital letter, all following `Theme.`. So, for example, filling in `AppTheme` will result in a style resource named `Theme.Apptheme`.

For "Style compatibility", choose "Holo". For "Base theme", choose the base style you want:

- Light
- Dark
- Light with dark action bar

The next four options ("Action bar style", "Action bar texture", "Tab hairline style", "Neutral pressed states") are for advanced features and can be left at their defaults.

Scrolling further down the page, you will come to seven color pickers, allowing you to tailor the colors to be used in your action bar implementation. Each picker, when opened, allows you to choose a color based on a fixed palette, then refined using a gradient selector. Or, if you know specific colors (e.g., a graphic designer gave them to you), you can fill the color into the supplied field:

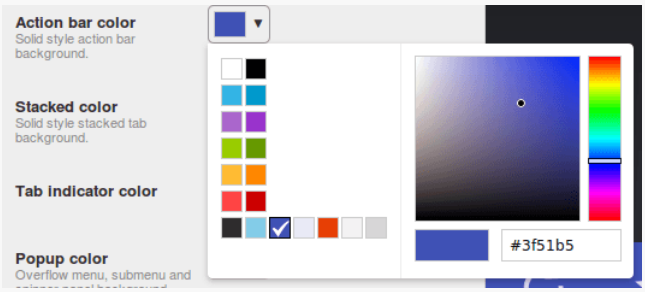


Figure 216: Action Bar Style Generator, Showing "Action bar color" Picker

As you change the colors, you will see what they impact on the preview.

At the bottom of the page is the "Output resources" frame:

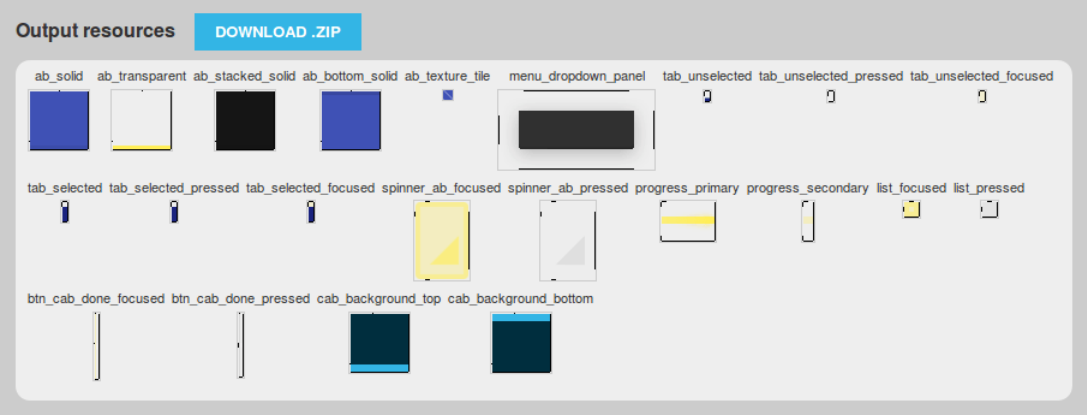


Figure 217: Action Bar Style Generator, Showing "Output resources" Frame

Here, you can click on the "DOWNLOAD .ZIP" button to download a ZIP archive containing your custom theme and all the associated resources required to implement it.

Implementing the Scheme

UnZIP the contents of that ZIP archive into your project's `res/` directory (e.g., in a traditional Android Studio project, unZIP into `src/main/res/` in your `app` module). It will add a bunch of files, notably including a file in `res/values/` whose name is based upon the name you filled into the Web form for the theme name (e.g., `styles_apptheme.xml`).

If you look at that file, you will see that it defines a custom theme for you, named `Theme.` plus whatever you provided to that form (converted into a leading capital letter and the rest lowercase). That file will be rather lengthy, as it designates specific styles to use for various facets of the action bar (e.g., `android:actionBarStyle`).

Here is the theme's primary `<style>` resource element, defining the theme itself:

```
<style name="Theme.Apptheme" parent="@android:style/Theme.Holo">
    <item name="android:actionBarItemBackground">@drawable/selectable_background_apptheme</item>
    <item name="android:popupMenuStyle">@style/PopupMenu.Apptheme</item>
    <item name="android:dropDownListViewStyle">@style/DropDownListView.Apptheme</item>
    <item name="android:actionBarTabStyle">@style/ActionBarTabStyle.Apptheme</item>
    <item name="android:actionDropDownStyle">@style/DropDownNav.Apptheme</item>
    <item name="android:actionBarStyle">@style/ActionBar.Solid.Apptheme</item>
    <item name="android:actionModeBackground">@drawable/cab_background_top_apptheme</item>
    <item name="android:actionModeSplitBackground">@drawable/cab_background_bottom_apptheme</item>
    <item name="android:actionModeCloseButtonStyle">@style/ActionButton.CloseMode.Apptheme</item>
```

```
  </style>
```

(from ActionBar/HoloColor/app/src/main/res/values/styles_apptheme.xml)

To use this theme, just add an `android:theme` attribute to your `<application>` (or perhaps individual `<activity>` elements) in your manifest:

```
<application
  android:allowBackup="false"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/Theme.Apptheme">
  <activity
    android:name="ActionBarDemoActivity"
    android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>

      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
</application>
```

(from ActionBar/HoloColor/app/src/main/AndroidManifest.xml)

The resulting app will have a color scheme mirroring what you defined on the form:

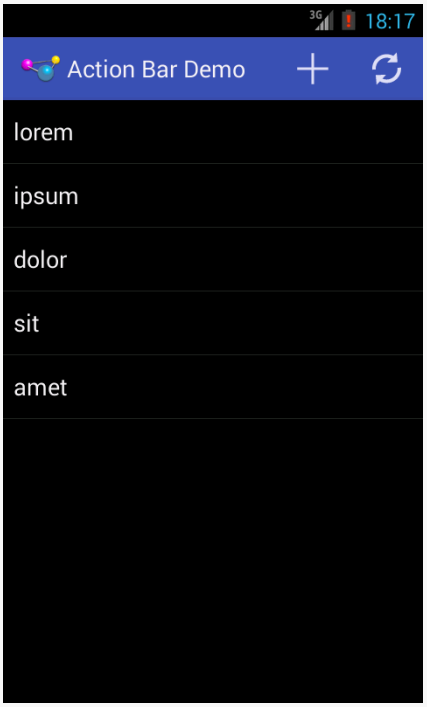


Figure 218: Results of the Action Bar Style Generator

This screenshot, and the code snippets, comes from the `ActionBar/HoloColor` sample project, which is the same as the base action bar sample app from this chapter with the custom theme applied.

Visit the Trails!

In addition to this chapter, you can learn more about advanced action bar techniques and learn about action modes, which temporarily replace the action bar with new items for use with contextual operations.

Vector Drawables

Android 5.0 added native support for a `VectorDrawable`, which uses the SVG path specification to represent vector art. However, unless your `minSdkVersion` was 21 or higher, vector drawable resources were not that useful, as there was no good way to support the same artwork on older devices. You could somehow arrange to have PNGs for the same artwork, but then, why bother with the vector artwork in the first place?

Nowadays, vector drawable resources are more practical. Not only do more devices run Android 5.0+, but we have better tool support. Android Studio offers a Vector Asset wizard that helps you add vector drawable resources to your project, and the build system can automatically generate PNG files at various densities to be used on older devices.

As a result, vector drawables have been gaining in popularity, particularly for action bar icons.

Getting the Artwork

You have two major sources of vector drawable artwork: XML files already in the vector drawable XML format, or SVG files that you wish to convert to vector drawable XML format. Since writing the vector drawable XML by hand will be difficult at best, most vector drawable XML will start from an SVG file. Whether you do the conversion, or whether somebody else did the conversion for you, is the major difference.

For SVG that you wish to try to convert to vector drawable XML, the simpler the SVG is, the more likely it is that you will have success. In particular, SVG features like gradients, patterns, and text are not supported. The apparent vision is for vector

drawable artwork to be used mostly for things like action bar icons, where things like gradients and patterns are not necessary.

Android Studio Vector Asset Wizard

The primary way most developers will get vector drawable XML into their projects is via the Android Studio Vector Asset wizard. You can bring this up by right-clicking over the `res/` directory of your desired source set, and choosing New > Vector Asset from the context menu:

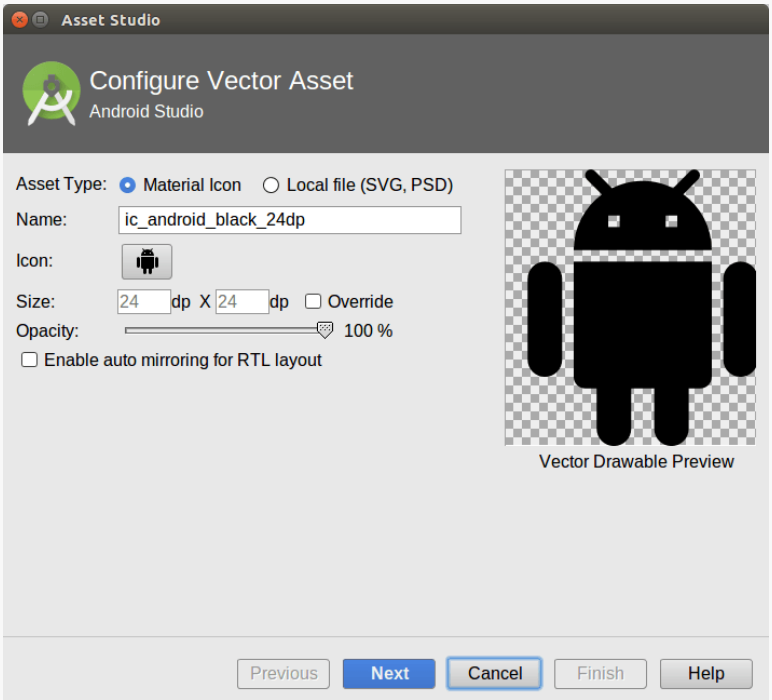


Figure 219: Android Studio Vector Asset Wizard

The "Asset Type" radio group gives you two sources of imagery: a subset of the official Material Design icons, or your own SVG or PSD file.

By default, the Material Icon radio button is selected. You can choose which icon to display by tapping the "Icon" button, which by default shows a rendition of the Android mascot. Tapping that button brings up a grid of icons for you to choose from:

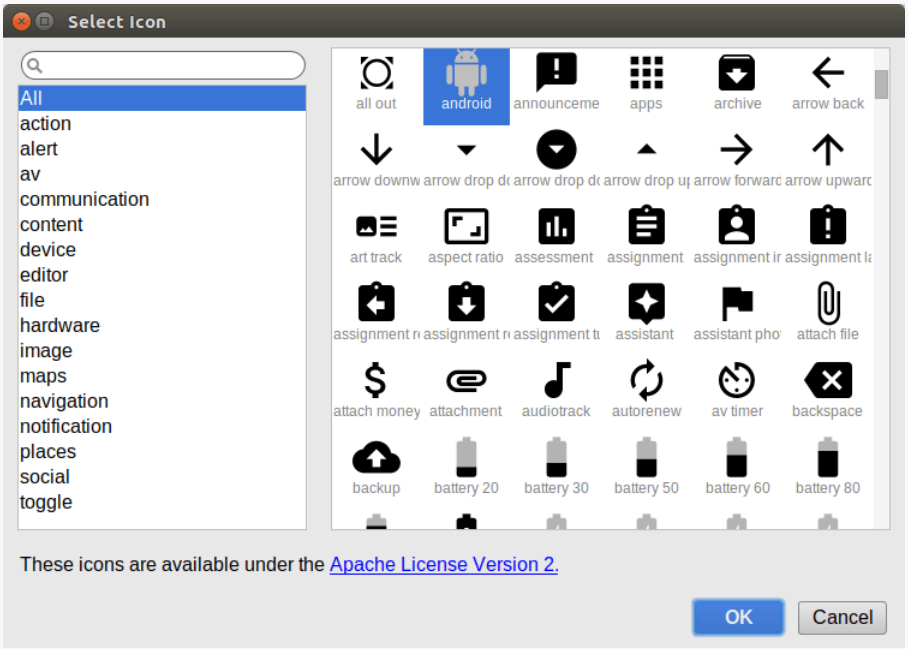


Figure 220: Android Studio Vector Asset Wizard, Material Icon Selector

You can browse by category or search by name to try to find the icon that you want from the library of available icons.

If you switch to the "Local file" radio button, the "Icon" button is replaced by a "Path" field, where you can pick the file that you wish to use.

By default, the Vector Asset wizard is trying to make action bar icon-sized images, 24dp square. You can override this by checking the "Override" checkbox and specifying your own size. The opacity slider allows you to indicate whether non-transparent pixels should be translucent (value from 0-99) or solid (100). If the image contains text or otherwise needs to be inverted for RTL languages, there is a checkbox to enable auto-mirroring support for that.

Also note that you can define the resource name, below where you chose the icon or SVG/PSD file. When importing a file, by default, the resource name will be the same as the base name of the file.

Clicking the "Next" button brings up a confirmation screen, where you can also change the module and source set if you perhaps brought up the wizard in the wrong spot:

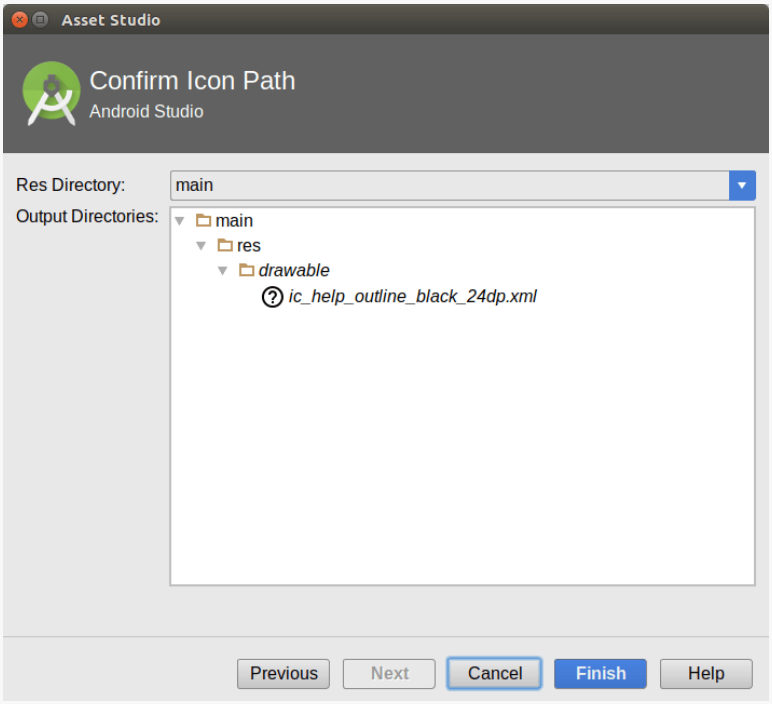


Figure 221: Android Studio Vector Asset Wizard, Confirmation Screen

Clicking Finish will import the resource and add it to `res/drawable/` in your project. When you build your project, if your `minSdkVersion` is below 21, the Android Gradle Plugin will generate PNG files to be used for those older devices. Note that these generated PNG files show up in your `build/` tree, not as part of your project source code.

The preview shown in the wizard should give you an indication if your SVG is being imported properly:

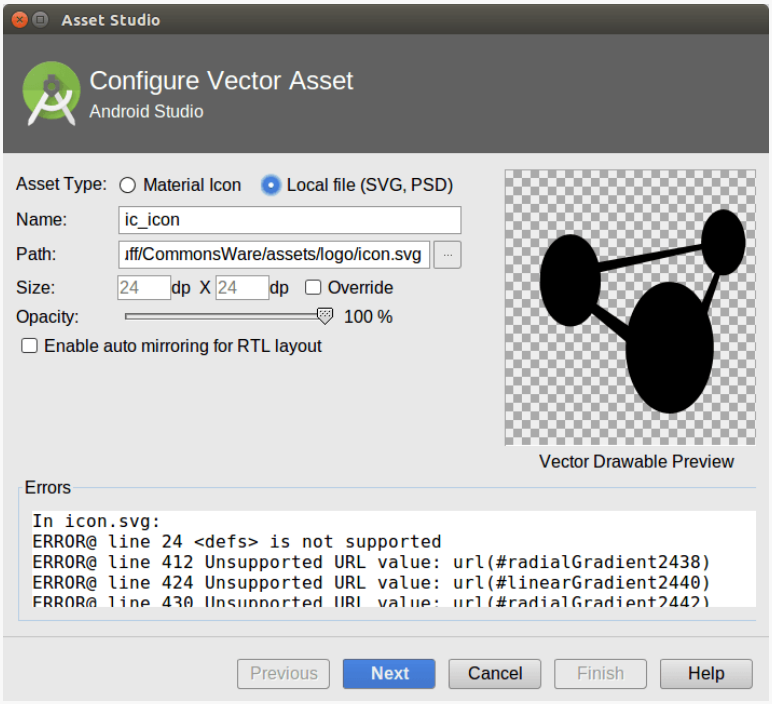


Figure 222: Android Studio Vector Asset Wizard, Showing Failed SVG Import

However, even if the preview turned out OK, be sure to test your app, both on Android 5.0+ and (if relevant) Android 4.4-and-older devices, to ensure that your artwork looks the way you want it to.

Other Tools

Juraj Novák maintains a separate Android SVG to vector drawable XML converter as a Web page. If you are running into problems with the Vector Asset wizard's import support, you might consider trying this site. It may give you better vector drawables directly, and it definitely gives you more indications about *why* your SVG may not convert properly.

Using the Artwork

You use vector drawable resources the same way that you use any other drawable resource. Under the covers, the Java class that handles rendering the artwork is `VectorDrawable`... on Android 5.0+.

If your `minSdkVersion` is below 21, and you want to use generated PNG files for the older devices, you need to add a line to the `defaultConfig` closure in your `android` closure in your module's `build.gradle` file:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 24
    buildToolsVersion "24.0.1"

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 24
        vectorDrawables.generatedDensities = ['hdpi','xxhdpi']
    }
}
```

Specifically, you need to set the `generatedDensities` property on the `vectorDrawables` object to an array of strings, identifying the densities for which you want PNGs generated. As with other drawable resources, devices operating on other densities than those in your chosen list will re-sample icons from one of your provided densities. If you have few vector drawables, you could list more densities and not consume much APK space. The sample shown above settles for two: `hdpi` for mid-range devices and `xxhdpi` for high-end devices.

If your `minSdkVersion` is 21 or higher, though, you do not need the generated PNG files, as all devices that will run your app will be capable of using the vector drawables natively.

VectorDrawableCompat

In February 2016, Google released `support-vector-drawable`. This contains a `VectorDrawableCompat` class that supports vector drawables going back to API Level 7. Google also released `animated-vector-drawable`, which offers `AnimatedVectorDrawableCompat`, supported back to API Level 11. Getting these going is tricky because they are largely undocumented and have significant limitations.

The `Drawable/Vector` sample project demonstrates the use of `VectorDrawableCompat`. In its `res/drawables-nodpi/` directory, you will find a handful of vector drawable resources, culled from the Android Open Source Project. The sample app will show those in a pair of `ListView` widgets in tabs:

- One will use `VectorDrawableCompat` and will work for all API levels that the project supports (15 and higher, based on the project's `minSdkVersion`).
- One will use native vector drawable support. As you will see shortly, we are going to disable the normal PNG generation from the vector drawables, which means that this list will only show the icons on Android 5.0 and higher, not older devices.

Gradle Configuration

To make `VectorDrawableCompat` work, you need to make some changes to your Gradle build files:

- You need to be using at least version 1.5.0 of the Android Plugin for Gradle, as defined in your project's top-level `build.gradle` file
- You need to have a dependency on `com.android.support:support-vector-drawable` in your module's `build.gradle` file, akin to your other Android Support dependencies

If you are using a 1.5.x version of the Android Gradle Plugin:

- You need to configure `generatedDensities` in your `defaultConfig` closure, inside your `android` closure in your module's `build.gradle` file
- You need `additionalParameters "--no-version-vectors"` in an `aaptOptions` closure inside your `android` closure in your module's `build.gradle` file

Whereas if you are using 2.0.0 or higher of the Android Plugin for Gradle, you need `vectorDrawables.useSupportLibrary = true` in the `defaultConfig` closure of your `android` closure in your module's `build.gradle` file.

These latter steps disable the automatic generation of PNG files from the vector drawable resources that would ordinarily happen by default.

The sample project uses version 1.5.0 of the Android Gradle Plugin, and therefore it has the `generatedDensities` and the `additionalParameters "--no-version-vectors"` statements:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'io.karim:materialtabs:2.0.2'
    implementation 'com.android.support:support-v13:23.2.0'
```

```
    implementation 'com.android.support:support-vector-drawable:23.2.0'
}

android {
    compileSdkVersion 23
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 22
        generatedDensities=[]
    }

    aaptOptions {
        additionalParameters "--no-version-vectors"
    }
}
```

(from Drawable/Vector/app/build.gradle)

Using a newer Android Gradle Plugin — 2.0.0 or higher – your `android` closure would have:

```
android {
   defaultConfig {
     vectorDrawables.useSupportLibrary = true
    }
}
```

Not generating PNG files saves us some disk space. On the other hand, now we are locked into using the vector drawable backport for these icons to be usable on older devices.

Use in Java

The sample app is cloned from another book sample, one that has a `ViewPager` with tabs. The activity simply sets up the `ViewPager` and tabs, using a `SampleAdapter` for the `ViewPager` contents. `SampleAdapter`, in turn, loads a `VectorFragment` or `VectorCompatFragment` into those tabs.

`VectorFragment` shows each of the icons in a row of a `ListView`, along with the resource name:

```
package com.commonsware.android.vector;
```

```
import android.app.ListFragment;
import android.os.Build;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;

public class VectorFragment extends ListFragment {
  private static final Integer[] VECTORS={
    R.drawable.ic_account_circle,
    R.drawable.ic_check_circle_24px,
    R.drawable.ic_corp_badge,
    R.drawable.ic_corp_icon_badge,
    R.drawable.ic_corp_statusbar_icon,
    R.drawable.ic_eject_24dp,
    R.drawable.ic_expand_more_48dp,
    R.drawable.ic_folder_24dp,
    R.drawable.ic_more_items,
    R.drawable.ic_perm_device_info,
    R.drawable.ic_sd_card_48dp,
    R.drawable.ic_settings_24dp,
    R.drawable.ic_storage_48dp,
    R.drawable.ic_usb_48dp
  };

  @Override
  public void onViewCreated(View view,
                            Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    setListAdapter(new VectorAdapter());
  }

  void applyIcon(ImageView icon, int resourceId) {
    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
      icon.setImageResource(resourceId);
    }
  }

  class VectorAdapter extends ArrayAdapter<Integer> {
    VectorAdapter() {
      super(getActivity(), R.layout.row, R.id.title, VECTORS);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
```

```
      View row=super.getView(position, convertView, parent);
      ImageView icon=(ImageView)row.findViewById(R.id.icon);
      TextView title=(TextView)row.findViewById(R.id.title);

      applyIcon(icon, getItem(position));
      title.setText(getResources().getResourceName(getItem(position)));

      return(row);
    }
  }
}
```

(from Drawable/Vector/app/src/main/java/com/commonsware/android/vector/VectorFragment.java)

Specifically:

- Our array is a roster of the drawable resource IDs, named `VECTORS`
- `VectorFragment` uses a `VectorAdapter` to populate the `ListView`
- `VectorAdapter`, in `getView()`, uses `getResourceName()` on a `Resources` object to get the resource name associated with a resource ID, to show in a `TextView` in the row
- `VectorAdapter` delegates to `VectorFragment` and its `applyIcon()` method to populate the `ImageView` given a drawable resource ID

The `VectorFragment` implementation of `applyIcon()` simply calls `setImageResource()` on the `ImageView`, supplying the drawable resource ID. This works fine on Android 5.0 and higher, but it will fail on older devices, because older Android devices do not know natively about vector drawable resources. Hence, we only update the icon if we are on API Level 21 or higher.

So, on an Android 6.0 device, we get:

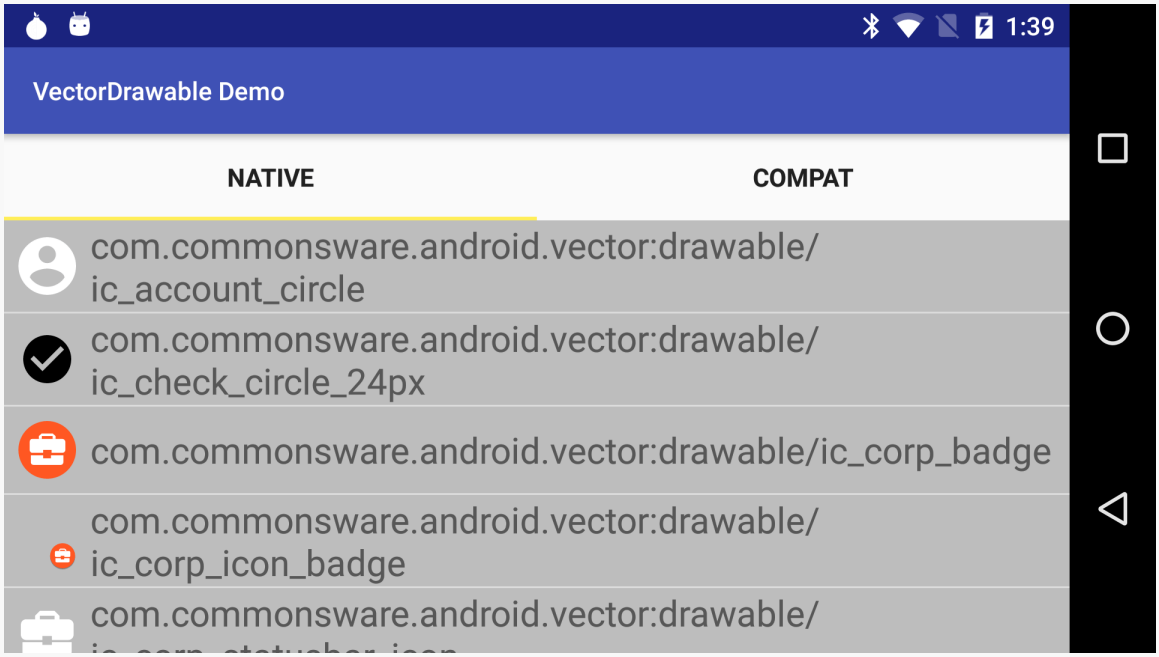


Figure 223: Vector Drawables, Native, Android 6.0

...while on an Android 4.4 device, we get:

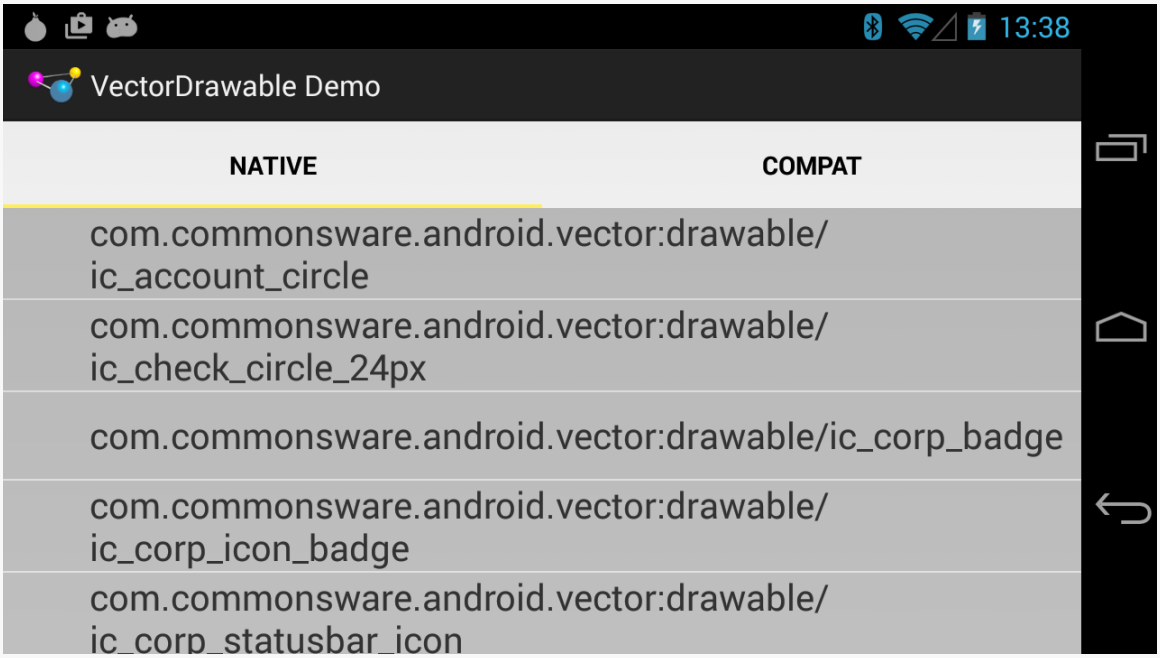


Figure 224: Vector Drawables, Missing on Android 4.4

`VectorCompatFragment` extends `VectorFragment` and simply overrides `applyIcon()`:

```
package com.commonsware.android.vector;

import android.graphics.drawable.Drawable;
import android.support.graphics.drawable.VectorDrawableCompat;
import android.widget.ImageView;

public class VectorCompatFragment extends VectorFragment {
  @Override
  void applyIcon(ImageView icon, int resourceId) {
    Drawable d=VectorDrawableCompat.create(getResources(),
      resourceId, null);

    icon.setImageDrawable(d);
  }
}
```

(from Drawable/Vector/app/src/main/java/com/commonsware/android/vector/VectorCompatFragment.java)

Here, we use `VectorDrawableCompat`, and its `static` `create()` method, to create a `Drawable` to apply to the `ImageView` via `setImageDrawable()`. `create()` takes three parameters:

- a `Resources` object
- a resource ID of a vector drawable
- an optional theme, or `null` to use the app's default theme

This approach works on all versions of Android supported by `VectorDrawableCompat`, which is API Level 7 and higher. However, on Android 5.0+ devices, `create()` will actually use a native vector drawable; the backport is only used on older devices.

The "Compat" tab of the Android 6.0 device and Android 4.4 device both show the icons:

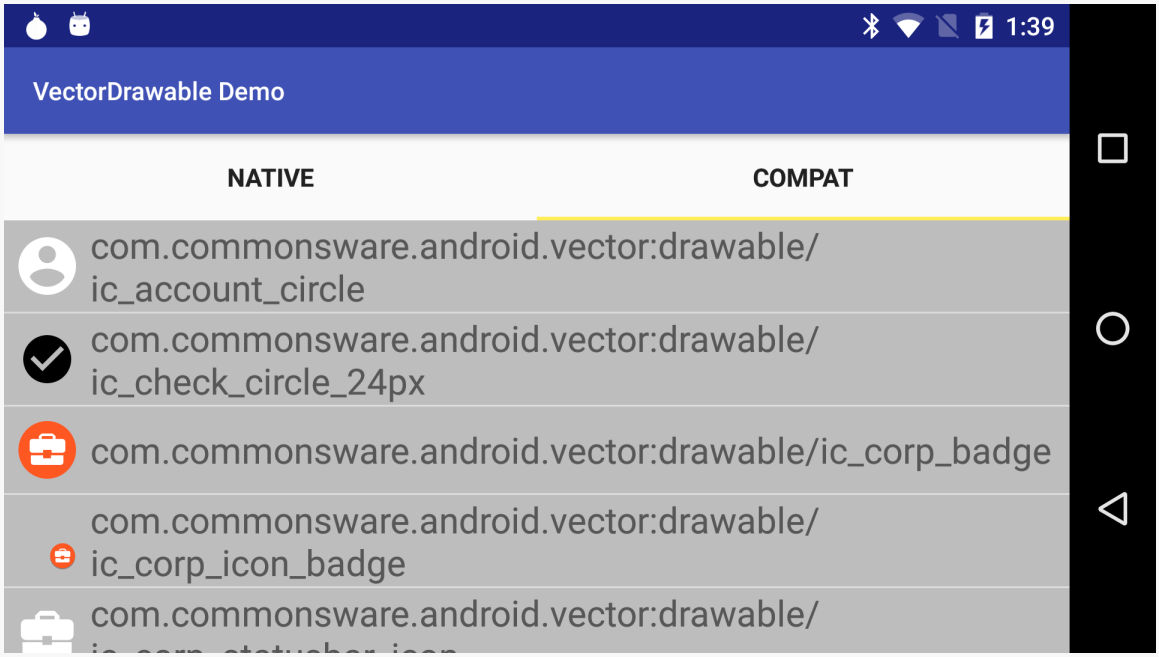


Figure 225: Vector Drawables, Loaded via VectorDrawableCompat, Android 6.0

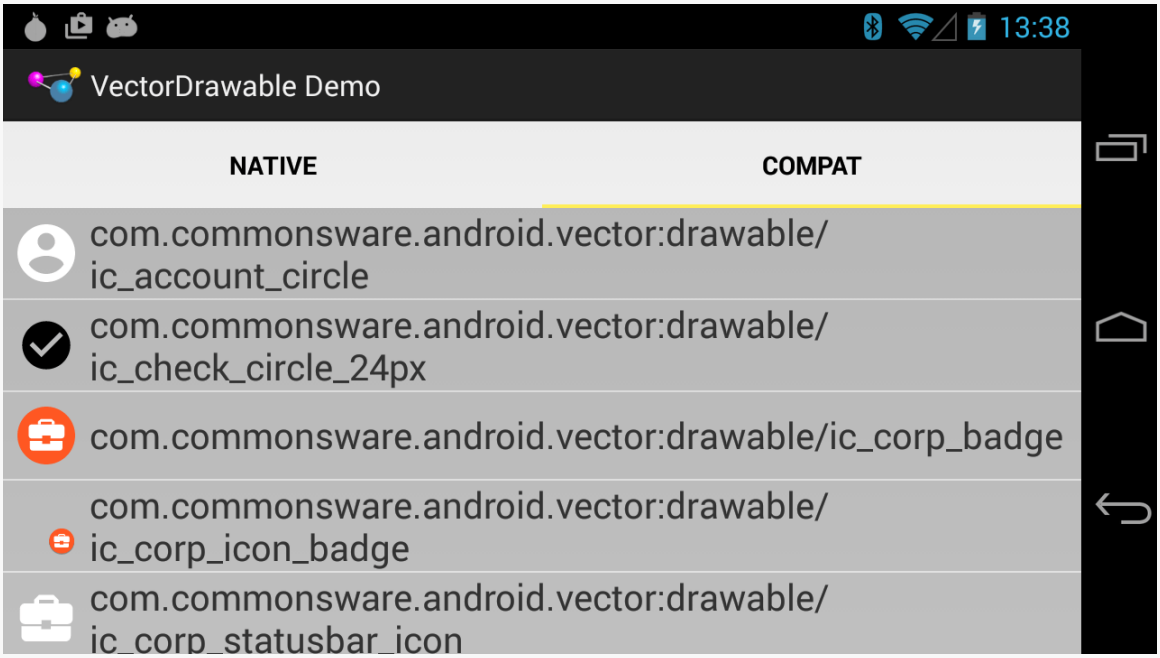


Figure 226: Vector Drawables, Loaded via VectorDrawableCompat, Android 4.4

Other VectorDrawable Backports

There are at least three independent backports, though, that you can try if you want to use the vector artwork directly on the older devices, rather than use PNGs generated from that vector artwork:

- https://github.com/wnafee/vector-compat
- https://github.com/a-student/BetterVectorDrawable
- https://github.com/telly/MrVector

Note that the last one was marked as deprecated, and the others may follow at some point, given that Google now has an official backport.

Tutorial #7 - Setting Up the Action Bar

Next up is to configure the action bar to our EmPubLite application.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Starting in this tutorial, we will now begin editing Java source files. Some useful Android Studio shortcut key combinations are:

- Alt-Enter (Option-Return on macOS) for bringing up quick-fixes for the problem at the code where the cursor is.
- Ctrl-Alt-O (Command-Option-O on macOS) will organize your Java `import` statements, including removing unused imports.
- Ctrl-Alt-L (Command-Option-L on macOS) will reformat the Java or XML in the current editing window, in accordance with either the default styles in Android Studio or whatever you have modified them to in Settings.

Step #1: Adding Some Icons

We are going to need a couple of icons for our action bar items. Nowadays, the preferred approach for doing this is to start with vector drawables.

Right-click over the `res/` directory and choose New > "Vector Asset" from the context menu. This brings up the first page of the vector asset wizard:

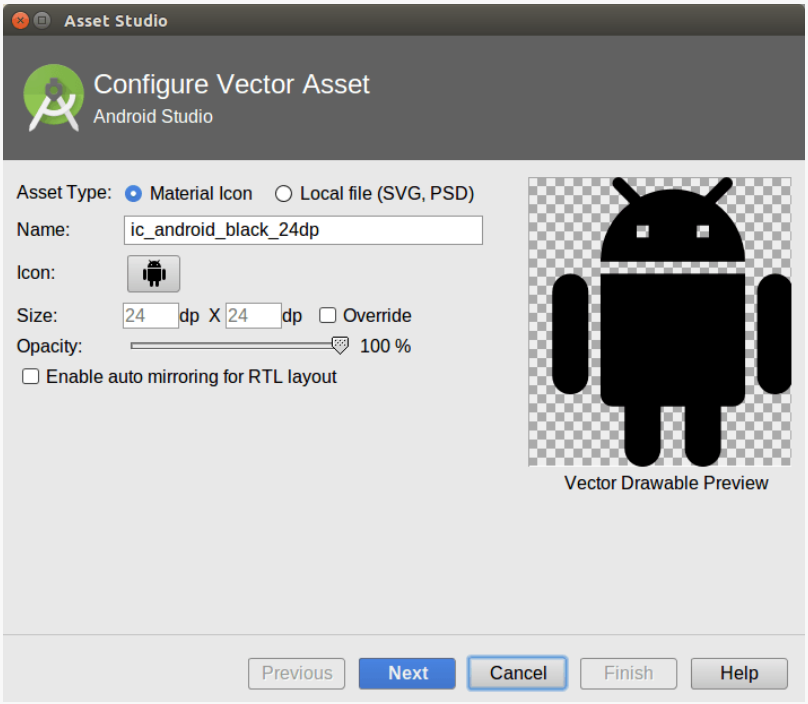


Figure 227: Android Studio Vector Asset Wizard, As Initially Launched

Click on the Icon button. This will bring up the material icon selector. In the search field, type `info`, then click on the "info outline" icon:

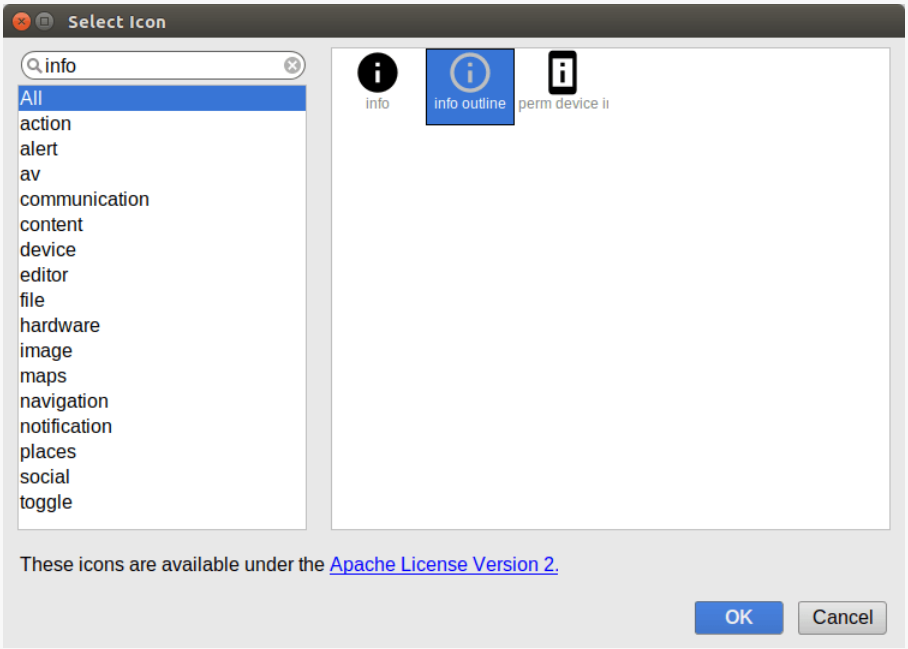


Figure 228: Android Studio Vector Asset Wizard, Material Icon Selector

Click "OK". This will update the name of the asset to `ic_info_outline_black_24dp`.

Click Next, then Finish, to add that icon as an XML file in `res/drawable/`.

Repeat that process to add a second vector asset, this time for "help outline" – you can search on `help` to quickly get to this icon.

Step #2: Defining Some Options

Next, we will add a couple of low-priority action items, for a help screen and an "about" screen.

Right click over the `res/` directory in your project, and choose New > "Android resource directory" from the context menu. This will bring up a dialog to let you create a new resource directory:

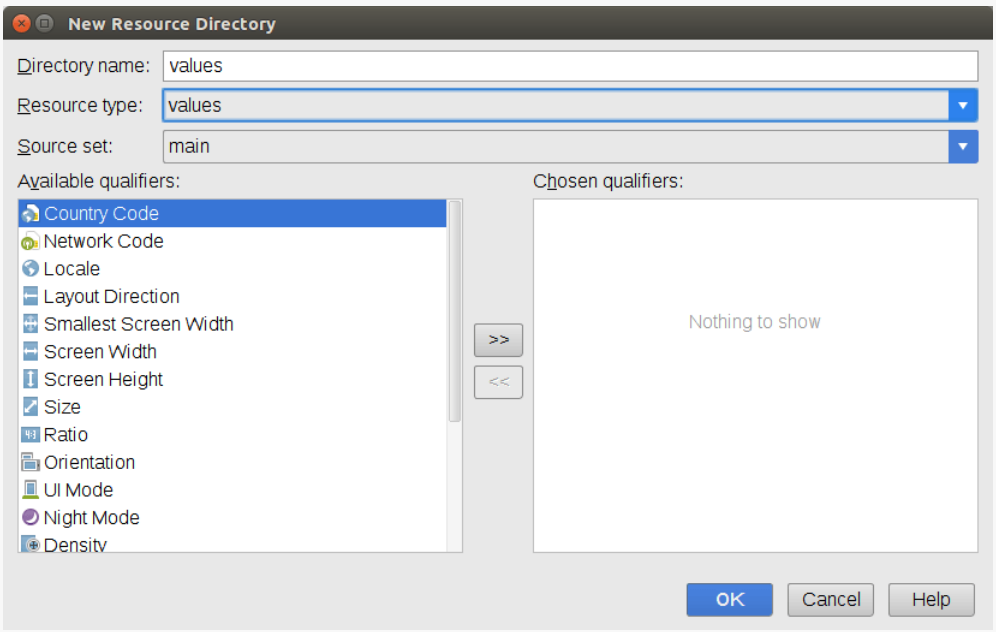


Figure 229: Android Studio New Resource Directory

Change the "Resource type" drop-down to be "menu", then click OK to create the directory.

Then, right-click over your new `res/menu/` directory and choose New > "Menu resource file" from the context menu. Fill in `options.xml` in the "New Menu Resource File" dialog:

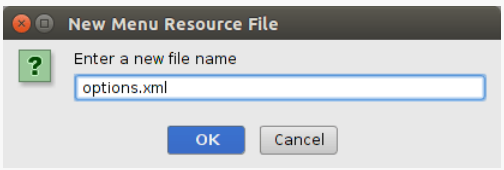


Figure 230: Android Studio New Menu Resource Dialog

Then click OK to create the file. It will open up into a menu editor:

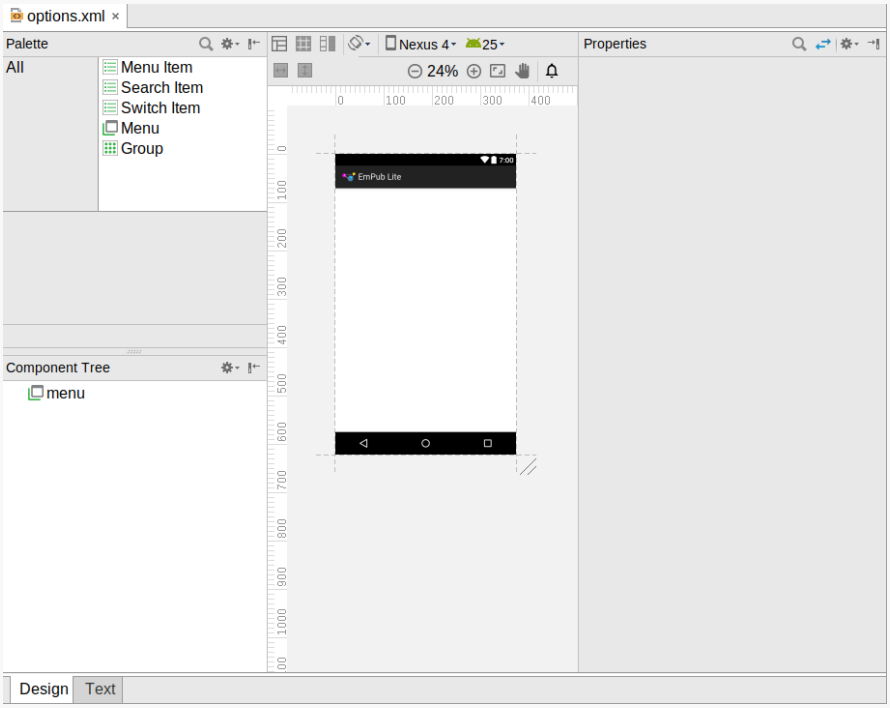


Figure 231: Android Studio Menu Resource Editor

Unfortunately, the drag-and-drop capabilities of this editor have many bugs. It will be simpler for you to switch to the Text sub-tab of the editor, into which you can paste the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/help"
    android:icon="@drawable/ic_help_outline_black_24dp"
    android:title="@string/help">
  </item>
  <item
    android:id="@+id/about"
    android:icon="@drawable/ic_info_outline_black_24dp"
    android:title="@string/about">
  </item>

</menu>
```

(from EmPubLite-AndroidStudio/T7-ActionBar/EmPubLite/app/src/main/res/menu/options.xml)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Also, you will need to add string resources for `help` and `about`, by adding appropriate `<string>` elements to your existing `res/values/strings.xml` file:

```
<resources>
  <string name="app_name">EmPub Lite</string>
  <string name="hint">Enter notes here</string>
  <string name="help">Help</string>
  <string name="about">About</string>
</resources>
```

(from EmPubLite-AndroidStudio/T7-ActionBar/EmPubLite/app/src/main/res/values/strings.xml)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #3: Loading and Responding to Our Options

Simply defining `res/menu/options.xml` is insufficient. We need to actually tell Android to use what we defined in that file, and we need to add code to respond to when the user taps on our items.

To do that, you will need to add an `onCreateOptionsMenu()` method and an `onOptionsItemSelected()` method to `EmPubLiteActivity`, as follows:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.about:
        return (true);

      case R.id.help:
        return (true);
    }
```

```
    return(super.onOptionsItemSelected(item));
  }
```

(from EmPubLite-AndroidStudio/T7-ActionBar/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

NOTE: Copying and pasting Java code may or may not work, depending on what you are using to read the book. For the PDF, some PDF viewers (e.g., Adobe Reader) should copy the code fairly well; others may do a much worse job. Reformatting the code with Ctrl-Alt-L (Command-Option-L on macOS) after pasting it in sometimes helps.

In `onCreateOptionsMenu()`, we are inflating `res/menu/options.xml` and pouring its contents into the supplied `Menu` object, which will be used by Android to populate our action bar.

In `onOptionsItemSelected()`, we examine the supplied `MenuItem` and route to different branches of a `switch` statement based upon the item's ID.

To get this to compile, you will need to add some imports as well:

```
import android.view.Menu;
import android.view.MenuItem;
```

(from EmPubLite-AndroidStudio/T7-ActionBar/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Android Studio users can press Alt-Enter (Command-Return on macOS) with the cursor in a class reference that is missing its import to add that import.

Step #4: Supporting Older Devices

As was noted in the previous chapter, for our vector drawables to work properly on older devices, we need to add a line to our `app/build.gradle` file, identifying the particular screen densities for which we want PNG editions of our vector drawables.

So, add a `vectorDrawables.generatedDensities` line to the `defaultConfig` closure, resulting in an `app/build.gradle` file that looks something like:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'
    defaultConfig {
        applicationId "com.commonsware.empublite"
```

```
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        vectorDrawables.generatedDensities = ['hdpi','xxhdpi']
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    androidTestImplementation('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    testImplementation 'junit:junit:4.12'
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation 'com.google.code.gson:gson:2.8.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
    implementation 'com.squareup.okhttp3:okhttp:3.4.1'
    implementation 'com.commonsware.cwac:security:0.8.0'
    implementation 'com.android.support:support-v13:25.3.0'
    implementation 'io.karim:materialtabs:2.0.5'
}
```

(from EmPubLite-AndroidStudio/T7-ActionBar/EmPubLite/app/build.gradle)

This will not have any immediate impact, as for Help and About, we are not actually using the icons. Those items are set to always be in the overflow. However, in later tutorials, we will add more action bar items, and some of those will be in the action bar proper and will have vector drawable icons. Furthermore, once you start using vector drawables, it is best to add the `generatedDensities` to your `build.gradle` file, so you are set once you really start using those vector drawables.

Step #5: Trying It Out

If you run this on an Android 4.4+ device or emulator, you should see a "..." icon on the action bar:

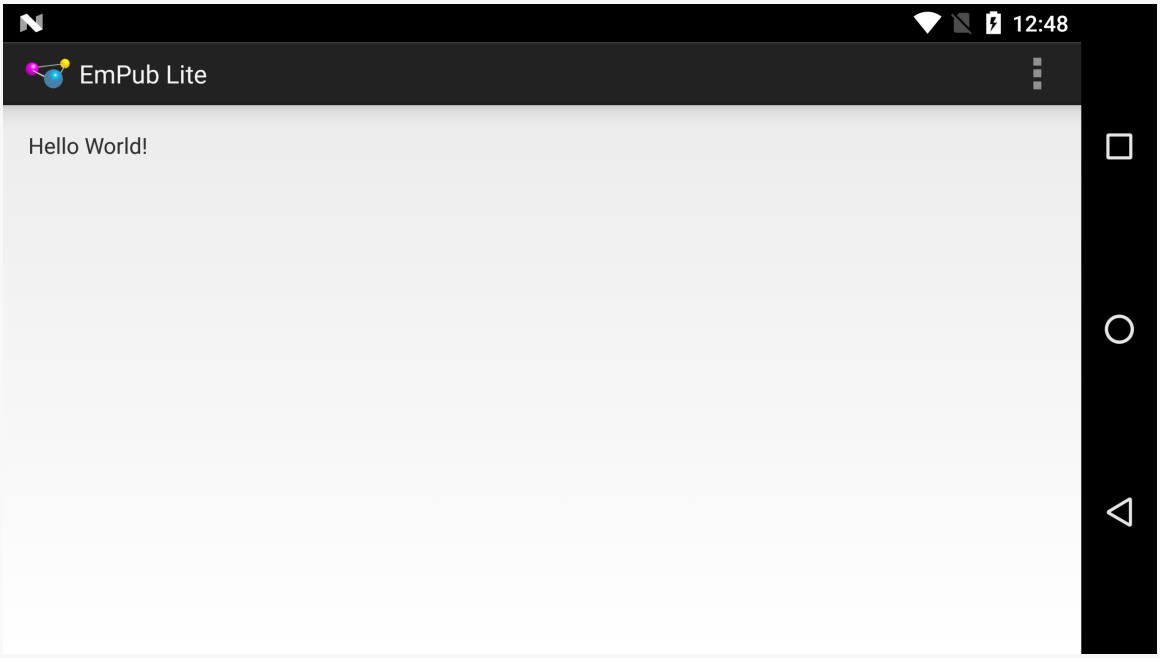


Figure 232: EmPubLite, Showing the ... Overflow Button

Pressing that brings up a menu showing our items:

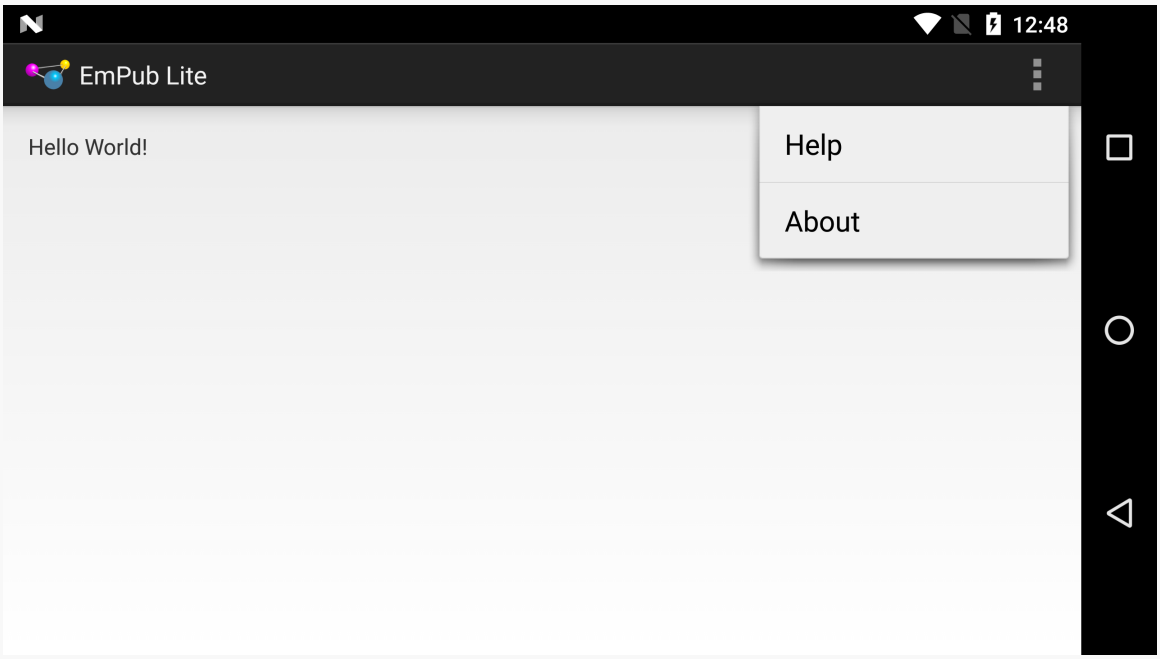


Figure 233: EmPubLite, Showing the Overflow Options

In Our Next Episode…

… we will define our first new activity on the tutorial project.

Android's Process Model

So far, we have been treating our activity like it is our entire application. Soon, we will start to get into more complex scenarios, involving multiple activities and other types of components, like services and content providers.

But, before we get into a lot of that, it is useful to understand how all of this ties into the actual OS itself. Android is based on Linux, and Linux applications run in OS processes. Understanding a bit about how Android and Linux processes inter-relate will be useful in understanding how our mixed bag of components work within these processes.

When Processes Are Created

A user installs your app, goes to their home screen's launcher, and taps on an icon representing your activity. Your activity dutifully appears on the screen.

Behind the scenes, what happened is that Android forked a copy of a process known as the `zygote`. As a result of the way your process is forked from the `zygote`, your process contains:

- A copy of the VM (Dalvik or ART), shared among all such processes via Linux copy-on-write memory sharing
- A copy of the Android framework classes, like `Activity` and `Button`, also shared via copy-on-write memory
- A copy of your own classes, loaded out of your APK
- Any objects created by you or the framework classes, such as the instance of your `Activity` subclass

BACK, HOME, and Your Process

Suppose that you have an app with just one activity. From the home screen's launcher, the user taps on the icon associated with your app's activity. Then, with your activity in the foreground, the user presses BACK.

At this point, the user is telling the OS that she is done with your activity. Control will return to whatever preceded that activity — in this case, the home screen's launcher.

You might think that this would cause your process to be terminated. After all, that is how most desktop operating systems work. Once the user closes the last window of the application, the process hosting that application is terminated.

However, that is not how Android works. Android will *keep* your process around, for a little while at least. This is done for speed and power: if the user happens to want to return to your app sooner rather than later, it is more efficient to simply bring up another copy of your activity again in the existing process than it is to go set up a completely new copy of the process. This does not mean that your process will live forever; we will discuss when your process will go away later in this chapter.

Now, instead of the user pressing BACK, let's say that the user pressed HOME instead. Visually, there is little difference: the home screen re-appears. Depending on the home screen implementation there may be a visible difference, as BACK might return to a launcher whereas HOME might return to something else on the home screen. However, in general, they feel like very similar operations.

The difference is what happens to your activity.

When the user presses BACK, your foreground activity is *destroyed*. We will get into more of what that means in the next chapter. However, the key feature is that the activity itself — the instance of your subclass of `Activity` – will never be used again, and hopefully is garbage collected.

When the user presses HOME, your foreground activity is *not* destroyed... at least, not immediately. It remains in memory. If the user launches your app again from the home screen launcher, and if your process is still around, Android will simply bring your existing activity instance back to the foreground, rather than having to create a brand-new one (as is the case if the user pressed BACK and destroyed your activity).

What HOME literally is doing is bringing the home screen activity back to the foreground, not otherwise directly affecting your process much.

Termination

Processes cannot live forever. They take up a chunk of RAM, for your classes and objects, and these mobile devices only have so much RAM to work with. Eventually, therefore, Android has to get rid of your process, to free up memory for other applications.

How long your process will stick around depends on a variety of factors, including:

- What else the device is doing, either in the foreground (user using apps) or in the background (e.g., automated checks for new email)
- How much memory the device has
- What is still running inside your process

Going back to the scenario from above, we have an application with a single activity launched from the home screen, where the user can return to the home screen either by pressing BACK or by pressing HOME. You might think that this makes no difference at all on when the process would be terminated, but that would be incorrect. Pressing HOME would keep the process around perhaps a bit longer than would pressing BACK.

Why?

When the user presses BACK, your one and only activity is destroyed. When the user presses HOME, your activity is not destroyed. Android will tend to keep processes around longer if they have active (i.e., not destroyed) components in them.

The key word there is "tend". Android's algorithms for determining when to get rid of what processes are baked into the OS and are, at best, lightly documented. There is evidence to suggest that other criteria, such as process age, are also taken into account, and so there may be times when a process that has an activity running (but not in the foreground) might be terminated where a process with no running activity might not. However, in general, processes with active (not destroyed) components will stick around a bit longer than processes without such components.

Foreground Means "I Love You"

Just because Android terminates processes to free up memory does not mean that it will terminate just any process to free up memory. A foreground process – the most common of which is a process that has an activity in the foreground – is the least likely of all to be terminated. In fact, you can pretty much assume that if Android has to kill off the foreground process, that the phone is very sick and will crash in a matter of moments.

(and, fortunately, that does not happen very often)

So, if you are in the foreground, you are safe. It is only when you are not in the foreground that you are at risk of having the process be terminated.

You and Your Heap

Processes take up RAM. A significant chunk of that RAM represents the objects you create (a.k.a., "the heap").

Those of you with significant Java backgrounds know that the Java VM loves RAM ("can't get enough of it!"). Java VMs routinely grab 64MB or 128MB of heap space upon creating the process and will grow as big as you wish to let them (e.g., `-Xmx` switch to the `java` command).

Android heap sizes are not that big, because Android is designed to run on mobile devices with constrained amounts of RAM.

Your heap limit may be as low as 16MB, though values in the 32-48MB range are more typical with current-generation devices. How much the heap limit will be depends a bit on what version of Android is on the device. It depends quite a lot, though, on the screen size, as bigger screens will tend to want to display bigger bitmap images, and bitmap images can consume quite a bit of RAM.

The key is that the heap is small, and (generally speaking) you cannot adjust it yourself. It is what it is. Small applications will rarely run into a problem with heap space, but larger applications might. We will discuss tools and techniques for measuring and coping with memory problems later in this book.

Activities and Their Lifecycles

An Android application will have multiple discrete UI facets. For example, a calendar application needs to allow the user to view the calendar, view details of a single event, edit an event (including adding a new one), and so forth. And on smaller-screen devices, like most phones, you may not have room to squeeze all of this on the screen at once.

To handle this, you can have multiple activities. Your calendar application may have one activity to display the calendar, another to add or edit an event, one to provide settings for how the calendar should work, another for your online help, etc. Some of these activities might be private to your app, while others might be able to be launched by third parties, such as your "launcher" activity being available to home screens.

All of this implies that one of your activities has the means to start up another activity. For example, if somebody clicks on an event from the view-calendar activity, you might want to show the view-event activity for that event. This means that, somehow, you need to be able to cause the view-event activity to launch and show a specific event (the one the user clicked upon).

This can be further broken down into two scenarios:

- You know what activity you want to launch, probably because it is another activity in your own application
- You have a reference to... something (e.g., a Web page), and you want your users to be able to do... something with it (e.g., view it), but you do not know up front what the options are

This chapter will cover both of those scenarios.

In addition, frequently it will be important for you to understand when activities are coming and going from the foreground, so you can automatically save or refresh data, etc. This is the so-called "activity lifecycle", and we will examine it in detail as well in this chapter.

Creating Your Second (and Third and…) Activity

Unfortunately, activities do not create themselves. On the positive side, this does help keep Android developers gainfully employed.

Hence, given a project with one activity, if you want a second activity, you will need to add it yourself. The same holds true for the third activity, the fourth activity, and so on.

The sample we will examine in this section is `Activities/Explicit`. Our first activity, `ExplicitIntentsDemoActivity`, started off as just the default activity code generated by the build tools. Now, though, its layout contains a `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:textSize="20sp"
    android:text="@string/hello"
    android:onClick="showOther"/>

</LinearLayout>
```

(from Activities/Explicit/app/src/main/res/layout/main.xml)

That `Button` is tied to a `showOther()` method in our activity implementation, which we will examine shortly.

Defining the Class and Resources

To create your second (or third or whatever) activity, you first need to create the Java class. You need to create a new Java source file, containing a public Java class that extends `Activity` directly or indirectly. You have two basic ways of doing this:

- Just create the class and resources yourself
- Use the Android Studio new-activity wizard

To use the Android Studio new-activity wizard, right-click on your `app/src/main/` source set directory in the project explorer, and go into the New > Activity portion of the context menu. This will give you a submenu of available activity templates — mostly the same roster of templates that we saw back when we created the project in the first place.

If you choose one of those templates, you will be presented with a one-page wizard in which to provide the details for this activity:

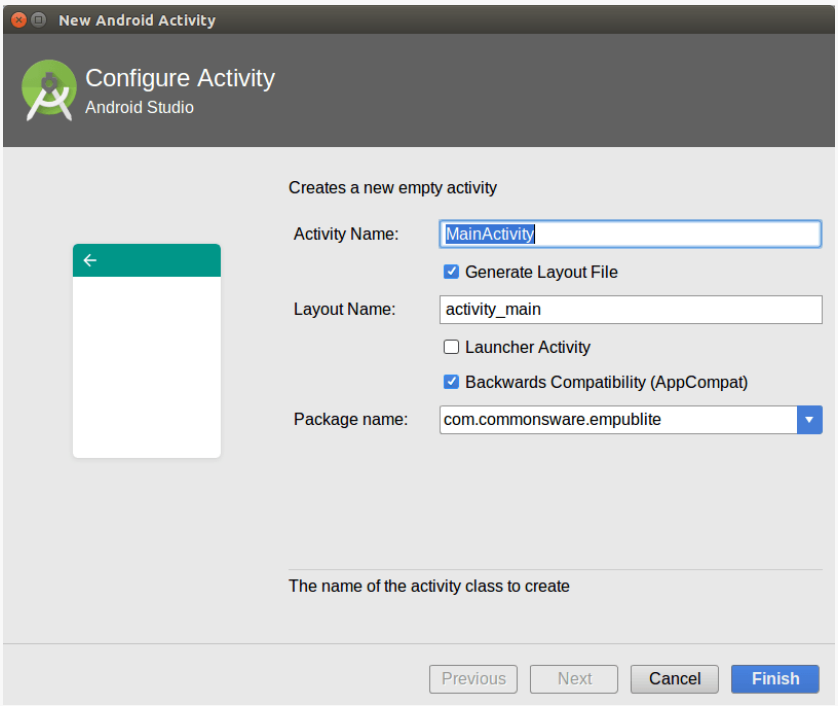


Figure 234: Android Studio New-Activity Wizard, Showing Empty Activity Template

What you see here will be based upon the template you chose (e.g., activity name, layout XML resource name) and will resemble those we saw back in the new-project wizard.

Clicking "Finish" will then create the activity's Java class, related resources (if any), and manifest entry.

Populating the Class and Resources

Once you have your stub activity set up, you can then add an `onCreate()` method to it (or edit an existing one created by the wizard), filling in all the details (e.g., `setContentView()`), just like you did with your first activity. Your new activity may need a new layout XML resource or other resources, which you would also have to create (or edit those created for you by the wizard).

In `Activities/Explicit`, our second activity is `OtherActivity`, with pretty much the standard bare-bones implementation:

```
package com.commonsware.android.exint;

import android.app.Activity;
import android.os.Bundle;

public class OtherActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.other);
  }
}
```

(from Activities/Explicit/app/src/main/java/com/commonsware/android/exint/OtherActivity.java)

and a similarly simple layout, `res/layout/other.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/other"
    android:textColor="#FFFF0000"
    android:textSize="20sp"/>

</LinearLayout>
```

(from Activities/Explicit/app/src/main/res/layout/other.xml)

Augmenting the Manifest

Simply having an activity implementation is not enough. We also need to add it to our `AndroidManifest.xml` file. This is automatically handled for you by the IDEs' respective new-activity wizards. However, if you created the activity "by hand", you will need to add its manifest element, and over time you will need to edit this element in many cases.

Adding an activity to the manifest is a matter of adding another `<activity>` element to the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.exint"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="ExplicitIntentsDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:name="OtherActivity"/>
  </application>

</manifest>
```

(from Activities/Explicit/app/src/main/AndroidManifest.xml)

You need the `android:name` attribute at minimum. Note that we do not include an `<intent-filter>` child element, as we did with the original activity has. For now, take it on faith that the original activity's `<intent-filter>` is what causes it to appear as a launchable activity in the home screen's launcher. We will get into more details of how that `<intent-filter>` works and when you might want your own in a later chapter.

Warning! Contains Explicit Intents!

An `Intent` encapsulates a request, made to Android, for some activity or other receiver to do something.

If the activity you intend to launch is one of your own, you may find it simplest to create an explicit `Intent`, naming the component you wish to launch. For example, from within your activity, you could create an `Intent` like this:

```
new Intent(this, HelpActivity.class);
```

This would stipulate that you wanted to launch the `HelpActivity`. `HelpActivity` would need to have a corresponding `<activity>` element in your `AndroidManifest.xml` file.

In `Activities/Explicit`, `ExplicitIntentsDemoActivity` has a `showOther()` method tied to its `Button` widget's `onClick` attribute. That method will use `startActivity()` with an explicit `Intent`, identifying `OtherActivity`:

```
package com.commonsware.android.exint;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ExplicitIntentsDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
    startActivity(new Intent(this, OtherActivity.class));
```

```
  }
}
```

(from Activities/Explicit/app/src/main/java/com/commonsware/android/exint/ExplicitIntentsDemoActivity.java)

Our launched activity shows the button:

Figure 235: The Explicit Intents Demo, As Launched

Clicking the button brings up the other activity:

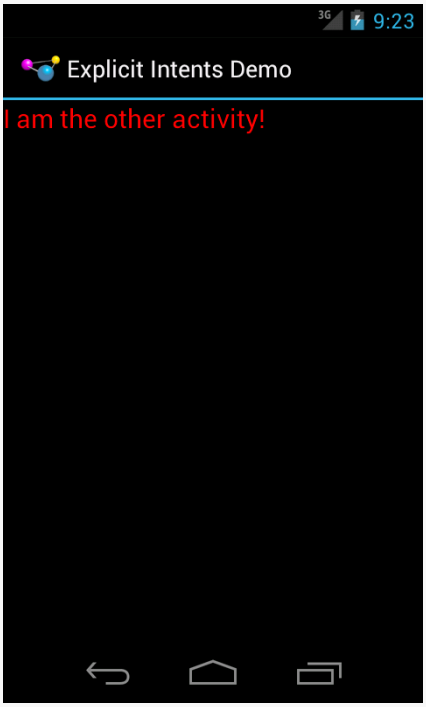


Figure 236: The Explicit Intents Demo, After Clicking the Button

Clicking BACK would return us to the first activity. In this respect, the BACK button in Android works much like the BACK button in your Web browser.

Using Implicit Intents

The explicit `Intent` approach works fine when the activity to be started is one of yours.

However, you can also start up activities from the operating system or third-party apps. In those cases, though, you will not have a Java `Class` object representing the other activity in your project, so you cannot use the `Intent` constructor that takes a `Class`.

Instead, you will use what are referred as the "implicit" `Intent` structure, which looks an *awful* lot like how the Web works.

If you have done any work on Web apps, you are aware that HTTP is based on verbs applied to URIs:

- We want to `GET` this image
- We want to `POST` to this script or controller
- We want to `PUT` to this REST resource
- Etc.

Android's implicit `Intent` model works much the same way, just with a *lot* more verbs.

For example, suppose you get a latitude and longitude from somewhere (e.g., body of a tweet, body of a text message). You decide that you want to display a map on those coordinates. There are ways that you can embed a Google Map directly in your app — and we will see how in a later chapter — but that is complicated and assumes the user wants Google Maps. It would be better if we could create some sort of generic "hey, Android, display an activity that shows a map for this location" request.

Or, in a simpler scenario: we get a URL to a Web page from some source (e.g., Web service call), and we want to open a Web browser on that page. This is illustrated in the `Activities/LaunchWeb` sample project.

We have a `LaunchDemo` activity that uses a layout containing a `EditText` widget and a `Button`, among other things:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <EditText
    android:id="@+id/url"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/url"
    android:inputType="textUri"/>

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me"/>

</LinearLayout>
```

(from Activities/LaunchWeb/app/src/main/res/layout/main.xml)

The `Button` is tied to a `showMe()` method on the activity itself, where we want to bring up a Web browser on the URL entered into the `EditText` widget:

```
package com.commonsware.android.activities;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class LaunchDemo extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
  }

  public void showMe(View v) {
    EditText url=(EditText)findViewById(R.id.url);

    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(url.getText().toString())));
  }
}
```

(from Activities/LaunchWeb/app/src/main/java/com/commonsware/android/activities/LaunchDemo.java)

Here, we take the URL and convert it to a `Uri` via calling `Uri.parse()`. Then, we can use an action called `ACTION_VIEW` to try to display the desired Web page.

When launched, the user is presented with our data entry form:

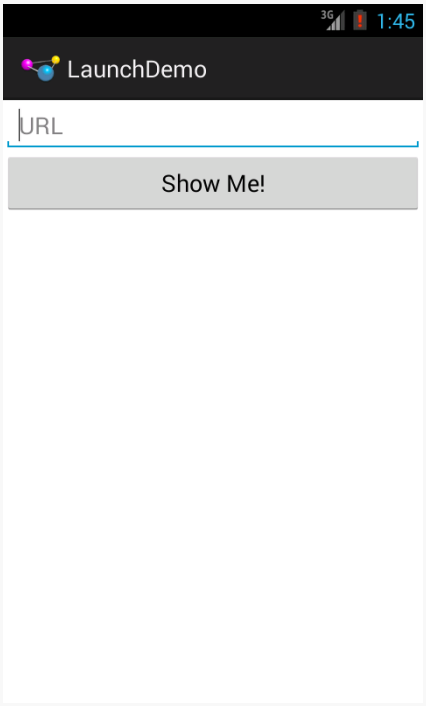


Figure 237: LaunchWeb Demo, As Initially Launched

We can fill in a URL:

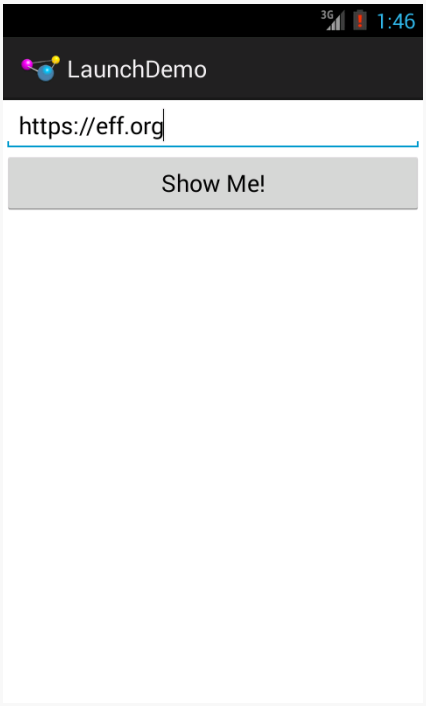


Figure 238: LaunchWeb Demo, After Data Entry

If the device has one app that responds to an `ACTION_VIEW` `Intent` on an `https:` scheme, clicking the "Show Me!" button will bring up that app, probably a Web browser:

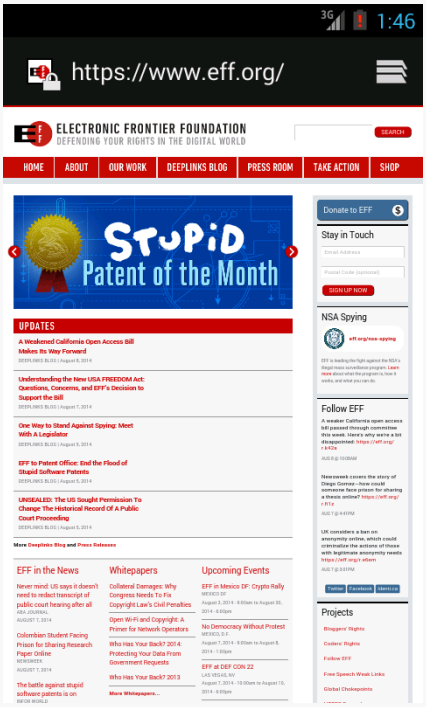


Figure 239: EFF Home Page, Launched from LaunchWeb

We will discuss what happens if there are no applications set up to handle this `Intent`, or if there is more than one, in a later chapter.

Where Do We Get These Uri Values?

In this example, we used `Uri.parse()` to parse an `https` URL. This is a typical approach for such URLs.

However, sometimes, what we want to view is on the device already, rather than being online. In those cases, we do not use `https`, or even `http`. Instead, we will use two other schemes: `file` and `content`.

The `file` scheme works more or less as it does with Web browsers. `file://` plus a path is a URL pointing to a file on the filesystem. We will get into working with files a bit later in the book. Given a `File` object, we can get the corresponding `Uri` via `Uri.fromFile()`.

The `content` scheme is for a `ContentProvider`. This is one of our Android components (along with activities, services, and broadcast receivers). A `content` `Uri` points to some content from that provider: a contact, a stream, a calendar entry, etc.

Usually, we get these as `Uri` values directly. If, somehow, you wind up with a `String` representation of a `content` `Uri`, `Uri.parse()` can turn that back into a `Uri`. We will get more into how to use a `ContentProvider` later in the book.

Extra! Extra!

Sometimes, we may wish to pass some data from one activity to the next. For example, we might have a `ListActivity` showing a collection of our model objects (e.g., books) and we have a separate `DetailActivity` to show information about a specific model object. Somehow, `DetailActivity` needs to know which model object to show.

One way to accomplish this is via `Intent` extras.

There is a series of `putExtra()` methods on `Intent` to allow you to supply key/value pairs of data to be bundled into the `Intent`. While you cannot pass arbitrary objects, most primitive data types are supported, as are strings and some types of lists. The next section will explain a bit more about what can go in an `Intent` extra.

Any activity can call `getIntent()` to retrieve the `Intent` used to start it up, and then can call various forms of `get... Extra()` (with the `...` indicating a data type) to retrieve any bundled extras.

For example, let's take a look at the `Activities/Extras` sample project.

This is mostly a clone of the `Activities/Explicit` sample from earlier in this chapter. However, this time, our first activity will pass an extra to the second:

```
package com.commonsware.android.extra;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ExtrasDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
```

```
    Intent other=new Intent(this, OtherActivity.class);

    other.putExtra(OtherActivity.EXTRA_MESSAGE, getString(R.string.other));
    startActivity(other);
  }
}
```

(from Activities/Extras/app/src/main/java/com/commonsware/android/extra/ExtrasDemoActivity.java)

We create the `Intent` as before, but then call `putExtra()`, supplying a key (a static string named `OtherActivity.EXTRA_MESSAGE`) and a value (the `R.string.other` string resource). Then, and only then, do we call `startActivity()`.

Our revised `OtherActivity` then retrieves that extra, along with the inflated `TextView` (via `findViewById()`) and pours that text in:

```
package com.commonsware.android.extra;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class OtherActivity extends Activity {
  public static final String EXTRA_MESSAGE="msg";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.other);

    TextView tv=(TextView)findViewById(R.id.msg);

    tv.setText(getIntent().getStringExtra(EXTRA_MESSAGE));
  }
}
```

(from Activities/Extras/app/src/main/java/com/commonsware/android/extra/OtherActivity.java)

Visually, the result is the same. Functionally, the text to be shown is passed from one activity to the next.

Pondering Parcelable

As noted above, `Intent` extras cannot handle arbitrary objects. That is because, most of the time, `Intent` extras get passed across process boundaries. Even when you are

calling `startActivity()` to start up one of your own activities, that request passes from your process to a core OS process and back to your process. The `Intent` extras come along for the ride.

Hence, `Intent` extras need to be something that can be converted into a `byte` array, as part of the inter-process communication (IPC) that handles the passing around of `Intent` objects. You will see this come up in other flavors of Android IPC as well, such as remote services.

However, there are two ways in which you can try to make your own objects work as `Intent` extras.

One approach is to implement `Serializable` on your class. This is a classic Java construct, designed to allow instances of your class, and other `Serializable` objects your instances hold onto, to be serialized into files and later read back in.

Another approach is to implement `Parcelable` on your class. This is an Android construct, one that is very similar to `Serializable`. However, `Serializable` is designed for *durable* storage of objects, where the file might be read back in months or years later. As such, `Serializable` has to deal with possible changes to the Java code implementing those classes, and as such needs to have hooks to help with converting old, saved objects into new objects. This adds overhead. `Parcelable` is only concerned with converting objects into `byte` arrays to pass across process boundaries. It can make the simplifying assumption that the class definition is not changing from when the object is turned into bytes and when the bytes are turned back into an object. As a result, `Parcelable` is faster than `Serializable` for Android's IPC use.

You are welcome to implement `Parcelable` on your own classes if you wish, at which point they can be passed around via `Intent` extras. Beyond that, though, any Java classes you see in the Android JavaDocs that implement `Parcelable` can be put into `Intent` extras. So, for example, `Uri` implements `Parcelable`, and so you can put a `Uri` into an `Intent` extra. Not everything in the Android SDK is `Parcelable`, but some key classes like `Uri` are `Parcelable`.

A lot more detail on `Parcelable`, including how you can implement it on your own classes, appears later in this book.

Asynchronicity and Results

Note that `startActivity()` is asynchronous. The other activity will not show up until sometime after you return control of the main application thread to Android.

Normally, this is not much of a problem. However, sometimes one activity might start another, where the first activity would like to know some "results" from the second. For example, the second activity might be some sort of "chooser", to allow the user to pick a file or contact or song or something, and the first activity needs to know what the user chose. With `startActivity()` being asynchronous, it is clear that we are not going to get that sort of result as a return value from `startActivity()` itself.

To handle this scenario, there is a separate `startActivityForResult()` method. While it too is asynchronous, it allows the newly-started activity to supply a result (via a `setResult()` method) that is delivered to the original activity via an `onActivityResult()` method. We will examine `startActivityForResult()` in greater detail in a later chapter.

Schroedinger's Activity

An activity, generally speaking, is in one of four states at any point in time:

1. *Active*: the activity was started by the user, is running, and is in the foreground. This is what you are used to thinking of in terms of your activity's operation.
2. *Paused*: the activity was started by the user, is running, and is visible, but another activity is overlaying part of the screen. During this time, the user can see your activity but may not be able to interact with it. This is a relatively uncommon state, as most activities are set to fill the screen, not have a theme that makes them look like some sort of dialog box.
3. *Stopped*: the activity was started by the user, is running, but it is hidden by other activities that have been launched or switched to.
4. *Dead*: the activity was destroyed, perhaps due to the user pressing the BACK button.

Life, Death, and Your Activity

Android will call into your activity as the activity transitions between the four states listed above.

Note that for all of these, you should chain upward and invoke the superclass' edition of the method, or Android may raise an exception.

`onCreate()` and `onDestroy()`

We have been implementing `onCreate()` in all of our `Activity` subclasses in all the examples. This will get called in two primary situations:

- When the activity is first started (e.g., since a system restart), `onCreate()` will be invoked with a `null` parameter.
- If the activity undergoes what we refer to as a "configuration change", such as the screen being rotated, by default your activity will be re-created and `onCreate()` will be called. We will discuss this scenario in greater detail later in this book.

Here is where you initialize your user interface and set up anything that needs to be done once, regardless of how the activity gets used.

On the other end of the lifecycle, `onDestroy()` may be called when the activity is shutting down, such as because the activity called `finish()` (which "finishes" the activity) or the user presses the BACK button. Hence, `onDestroy()` is mostly for cleanly releasing resources you obtained in `onCreate()` (if any), plus making sure that anything you started up outside of lifecycle methods gets stopped, such as background threads.

Bear in mind, though, that `onDestroy()` may not be called. This would occur in a few circumstances:

- You crash with an unhandled exception
- The user force-stops your application, such as through the Settings app
- Android has an urgent need to free up RAM (e.g., to handle an incoming phone call), wants to terminate your process, and cannot take the time to call all the lifecycle methods

Hence, `onDestroy()` is very likely to be called, but it is not guaranteed.

Also, bear in mind that it may take a long time for `onDestroy()` to be called. It is called quickly if the user presses BACK to finish the foreground activity. If, however, the user presses HOME to bring up the home screen, your activity is not immediately destroyed. `onDestroy()` will not be called until Android does decide to gracefully terminate your process, and that could be seconds, minutes, or hours later.

`onStart()`, `onRestart()`, and `onStop()`

An activity can come to the foreground either because it is first being launched, or because it is being brought back to the foreground after having been hidden (e.g., by another activity, by an incoming phone call).

The `onStart()` method is called in either of those cases. The `onRestart()` method is called in the case where the activity had been stopped and is now restarting.

Conversely, `onStop()` is called when the activity is about to be stopped. It too may not be called, for the same reasons that `onDestroy()` would not be called. However, `onStop()` is usually called fairly quickly after the activity is no longer visible, so the odds that `onStop()` will be called are even higher than that of `onDestroy()`.

`onPause()` and `onResume()`

The `onResume()` method is called just before your activity comes to the foreground, either after being initially launched, being restarted from a stopped state, or after a pop-up dialog (e.g., incoming call) is cleared. This is a great place to refresh the UI based on things that may have occurred since the user last was looking at your activity. For example, if you are polling a service for changes to some information (e.g., new entries for a feed), `onResume()` is a fine time to both refresh the current view and, if applicable, kick off a background thread to update the view (e.g., via a `Handler`).

Conversely, anything that takes over user input — mostly, the activation of another activity — will result in your `onPause()` being called. Here, you should undo anything you did in `onResume()`, such as stopping background threads, releasing any exclusive-access resources you may have acquired (e.g., camera), and the like.

Once `onPause()` is called, Android reserves the right to kill off your activity's process at any point. Hence, you should not be relying upon receiving any further events.

So, what is the difference between `onPause()` and `onStop()`? If an activity comes to the foreground that fills the screen, your current foreground activity will be called with `onPause()` and `onStop()`. If, however, an activity comes to the foreground that does *not* fill the screen, your current foreground activity will only be called with `onPause()`, as it is still visible.

Stick to the Pairs

If you initialize something in `onCreate()`, clean it up in `onDestroy()`.

If you initialize something in `onStart()`, clean it up in `onStop()`.

If you initialize something in `onResume()`, clean it up in `onPause()`.

In other words, stick to the pairs. For example, do not initialize something in `onStart()` and try to clean it up in `onPause()`, as there are scenarios where `onPause()` may be called multiple times in succession (i.e., user brings up a non-full-screen activity, which triggers `onPause()` but not `onStop()`, and hence not `onStart()`).

Which pairs of lifecycle methods you choose is up to you, depending upon your needs. You may decide that you need two pairs (e.g., `onCreate()`/`onDestroy()` and `onResume()`/`onPause()`). Just do not mix and match between them.

When Activities Die

So, what gets rid of an activity? What can trigger the chain of events that results in `onDestroy()` being called?

First and foremost, when the user presses the BACK button, the foreground activity will be destroyed, and control will return to the previous activity in the user's navigation flow (i.e., whatever activity they were on before the now-destroyed activity came to the foreground).

You can accomplish the same thing by calling `finish()` from your activity. This is mostly for cases where some other UI action would indicate that the user is done with the activity (e.g., the activity presents a list for the user to choose from — clicking on a list item might close the activity). However, please do not artificially add your own "exit", "quit", or other menu items or buttons to your activity — just allow the user to use normal Android navigation options, such as the BACK button.

If none of your activities are in the foreground any more, your application's process is a candidate to be terminated to free up RAM. As noted earlier, depending on circumstances, Android may or may not call `onDestroy()` in these cases (`onPause()` and `onStop()` would have been called when your activities left the foreground).

If the user causes the device to go through a "configuration change", such as switching between portrait and landscape, Android's default behavior is to destroy your current foreground activity and create a brand new one in its place. We will cover this more in a later chapter.

And, if your activity has an unhandled exception, your activity will be destroyed, though Android will not call any more lifecycle methods on it, as it assumes your activity is in an unstable state.

Walking Through the Lifecycle

To see when these various lifecycle methods get called, let's examine the `Activities/Lifecycle` sample project.

This project is the same as the `Activities/Extras` project, except that our two activities no longer inherit from `Activity` directly. Instead, we introduce a `LifecycleLoggingActivity` as a base class and have our activities inherit from it:

```
package com.commonsware.android.lifecycle;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class LifecycleLoggingActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Log.d(getClass().getSimpleName(), "onCreate()");
  }

  @Override
  public void onRestart() {
    super.onRestart();

    Log.d(getClass().getSimpleName(), "onRestart()");
  }
```

```
  @Override
  public void onStart() {
    super.onStart();

    Log.d(getClass().getSimpleName(), "onStart()");
  }

  @Override
  public void onResume() {
    super.onResume();

    Log.d(getClass().getSimpleName(), "onResume()");
  }

  @Override
  public void onPause() {
    Log.d(getClass().getSimpleName(), "onPause()");

    super.onPause();
  }

  @Override
  public void onStop() {
    Log.d(getClass().getSimpleName(), "onStop()");

    super.onStop();
  }

  @Override
  public void onDestroy() {
    Log.d(getClass().getSimpleName(), "onDestroy()");

    super.onDestroy();
  }
}
```

(from Activities/Lifecycle/app/src/main/java/com/commonsware/android/lifecycle/LifecycleLoggingActivity.java)

All `LifecycleLoggingActivity` does is override each of the lifecycle methods mentioned above and emit a debug line to Logcat indicating who called what.

When we first launch the application, our first batch of lifecycle methods is invoked, in the expected order:

```
04-01 11:47:21.437: D/ExplicitIntentsDemoActivity(1473): onCreate()
```

```
04-01 11:47:21.827: D/ExplicitIntentsDemoActivity(1473): onStart()
04-01 11:47:21.827: D/ExplicitIntentsDemoActivity(1473): onResume()
```

If we click the button on the first activity to start up the second, we get:

```
04-01 11:47:54.776: D/ExplicitIntentsDemoActivity(1473): onPause()
04-01 11:47:54.877: D/OtherActivity(1473): onCreate()
04-01 11:47:54.947: D/OtherActivity(1473): onStart()
04-01 11:47:54.974: D/OtherActivity(1473): onResume()
04-01 11:47:55.347: D/ExplicitIntentsDemoActivity(1473): onStop()
```

Notice that our first activity is paused before the second activity starts up, and that `onStop()` is delayed on the first activity until after the second activity has appeared.

If we press the BACK button on the second activity, returning to the first activity, we see:

```
04-01 11:48:54.807: D/OtherActivity(1473): onPause()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onRestart()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onStart()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onResume()
04-01 11:48:55.257: D/OtherActivity(1473): onStop()
04-01 11:48:55.257: D/OtherActivity(1473): onDestroy()
```

Notice how, once again, going onto the screen happens in between `onPause()` and `onStop()` of the activity leaving the screen. Also notice that `onDestroy()` is called immediately after `onStop()`, because the activity was finished via the BACK button.

If we now press the HOME button, to bring the home screen activity to the foreground, we see:

```
04-01 11:50:30.347: D/ExplicitIntentsDemoActivity(1473): onPause()
04-01 11:50:32.227: D/ExplicitIntentsDemoActivity(1473): onStop()
```

There is a delay between `onPause()` and `onStop()` as the home screen does its display work, and there is no `onDestroy()`, because the application is still running and nothing finished the activity. Eventually, the device will terminate our process, and if that happens normally, we would see the `onDestroy()` Logcat message.

Recycling Activities

Let us suppose that we have three activities, named A, B, and C. A starts up an instance of B based on some user input, and B later starts up an instance of C through some more user input.

Our "activity stack" is now A-B-C, meaning that if we press BACK from C, we return to B, and if we press BACK from B, we return to A.

Now, let's suppose that from C, we wish to navigate back to A. For example, perhaps the user pressed the icon on the left of our action bar, and we want to return to the "home activity" as a result, and in our case that happens to be A. If C calls `startActivity()`, specifying A, we wind up with an activity stack that is A-B-C-A.

That's because starting an activity, by default, creates a new instance of that activity. So, now we have two independent copies of A.

Sometimes, this is desired behavior. For example, we might have a single `ListActivity` that is being used to "drill down" through a hierarchical data set, like a directory tree. We might elect to keep starting instances of that same `ListActivity`, but with different extras, to show each level of that hierarchy. In this case, we would want independent instances of the activity, so the BACK button behaves as the user might expect.

However, when we navigate to the "home activity", we may not want a separate instance of A.

How to address this depends a bit on what you want the activity stack to look like after navigating to A.

If you want an activity stack that is B-C-A — so the existing copy of A is brought to the foreground, but the instances of B and C are left alone — then you can add `FLAG_ACTIVITY_REORDER_TO_FRONT` to your `Intent` used with `startActivity()`:

```
Intent i=new Intent(this, HomeActivity.class);

i.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);
startActivity(i);
```

If, instead, you want an activity stack that is just A — so if the user presses BACK, they exit your application — then you would add two flags: `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_SINGLE_TOP`:

```
Intent i=new Intent(this, HomeActivity.class);

i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_SINGLE_TOP);
startActivity(i);
```

This will finish all activities in the stack between the current activity and the one you are starting — in our case, finishing C and B.

Application: Transcending the Activity

`Activity` inherits from a class named `Context`. Many of the methods that we are calling on our activities, like `startActivity()`, are inherited from `Context`.

However, `Activity` is not the only relevant subclass of `Context`. We will see `Service` later in the book, for example. And sometimes we will see plain `Context` objects, such as when we cover `BroadcastReceiver` later in the book.

Another `Context` of note is `Application`. An instance of `Application` is created when our app starts up. The `Application` instance is a natural singleton; there should be exactly one instance of `Application` in our process.

Normally, this singleton is an instance of `Application` itself. However, we can subclass `Application` if we wish, then include a reference to our custom application class in an `android:name` attribute on the `<application>` element in the manifest. Then, when Android starts up our app, it will create an instance of our designated `Application` subclass, rather than creating an instance of the ordinary `Application` class.

We can retrieve the `Application` object at any point by calling `getApplicationContext()` on any `Context` object. `getApplicationContext()` will return a `Context`; if we need to reference `Application` or our specific `Application` subclass, we need to down-cast the returned `Context` to the appropriate type.

We can use `Application` in a few ways in Android apps.

First and foremost, if we need to hold onto some other object in a `static` data member, and that other object needs a `Context`, we *really* want it to be using the

`Application`, not an `Activity`, `Service`, etc. Because `Application` is a singleton, it is effectively "pre-leaked". We cannot somehow leak it further by having another indirect `static` reference to it. In contrast, suppose we have a `static` data member holding onto an `Activity`. Now, when that `Activity` is destroyed, it (and all it holds, like widgets and listeners) cannot be garbage-collected. This represents a memory leak.

You could even take it one step further and have the `Application` manage this static data, rather than using separate singletons. There are pros and cons to this approach, but on the whole Google is not a big fan of it. That being said, `Application` has an `onCreate()` that is called shortly after it is instantiated, and your subclass of `Application` could override that and use it to initialize some "global" data.

However, while the JavaDocs indicate that there is an `onTerminate()` method on `Application` — suggesting that we find out when the `Application` is going away and our process is being terminated — that method is never called in practice.

The Case of the Invisible Activity

Sometimes, you want an activity that has no UI.

This is rather unusual. Mostly, it will be cases where something else in the system says that *it* needs *you* to have an activity, but where you do not really have anything that you want to display to the user in a traditional activity-style UI.

For example, home screen launcher icons only start up activities. However, you may have a need for a home screen launcher that simply triggers some work to be done in the background, perhaps using a service (as will be discussed later in the book).

You have two ways of setting up an invisible activity, both involving using a particular `android:theme` value on the `<activity>` element.

The most efficient option is to use `Theme.Translucent.NoTitleBar`. This sets up your activity to have a transparent background and no action bar. The user may still perceive that the activity is around — for example, it will show up in the overview screen (a.k.a., recent-tasks list). Also, since the activity is "really there", the user may not be able to interact with whatever the user can see, such as the underlying home screen. But, if the activity can `finish()` itself quickly, and is interacting with the

user in the meantime (e.g., displaying some system dialog), you may be able to get away with this approach.

You will also see projects use a `Theme.NoDisplay` theme. This says that there should be no window associated with this particular activity. This too results in an invisible activity.

Occasionally, you need an invisible activity that has to hang around for a few seconds, perhaps waiting on some callback result, before it can be destroyed. Using `Theme.NoDisplay` will still work... but only on older Android devices. On Android 6.0 and higher, using `Theme.NoDisplay` without calling `finish()` in `onCreate()` (or, technically, before `onResume()`) will crash your app. This is why the recommendation is to use `Theme.Translucent.NoTitleBar`, which does not suffer from this limitation.

Tutorial #8 - Setting Up An Activity

Of course, it would be nice if those "Help" and "About" menu choices that we added in the previous tutorial actually did something.

In this tutorial, we will define another activity class, one that will be responsible for displaying simple content like our help text and "about" details. And, we will arrange to start up that activity when those action bar items are selected. The activity will not actually display anything meaningful yet, as that will be the subject of the next few tutorials.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Creating the Stub Activity Class and Manifest Entry

First, we need to define the Java class for our new activity, `SimpleContentActivity`.

Right-click on your `main/` source set directory in the project explorer, and choose New > Activity > Empty Activity from the context menu. This will bring up a new-activity wizard:

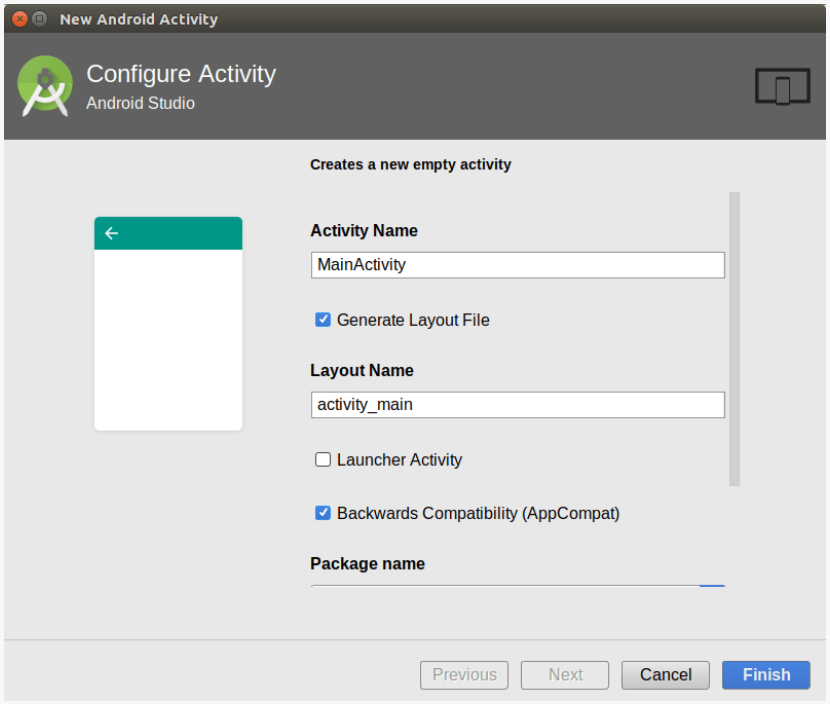


Figure 240: Android Studio New Activity Wizard

Fill in `SimpleContentActivity` in the "Activity Name" field and uncheck the "Generate Layout File" checkbox. Leave "Launcher Activity" unchecked, and uncheck the "Backwards Compatibility (AppCompat)" checkbox. If the package name drop-down is showing the app's package name, leave it alone. On the other hand, if the package name drop-down is empty, click on it and choose the app's package name. Leave the source language drop-down set to Java. Then click on Finish.

At this point, your `SimpleContentActivity` class should look like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;

public class SimpleContentActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }
}
```

(from EmPubLite-AndroidStudio/T8-Activities/EmPubLite/app/src/main/java/com/commonsware/empublite/SimpleContentActivity.java)

Step #2: Launching Our Activity

Now that we have declared that the activity exists and can be used, we can start using it.

Go into `EmPubLiteActivity` and modify `onOptionsItemSelected()` to add in some logic in the `R.id.about` and `R.id.help` branches, as shown below:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.about:
      Intent i=new Intent(this, SimpleContentActivity.class);
      startActivity(i);

      return(true);

    case R.id.help:
      i=new Intent(this, SimpleContentActivity.class);
      startActivity(i);

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from EmPubLite-AndroidStudio/T8-Activities/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

In those two branches, we create an `Intent`, pointing at our new `SimpleContentActivity`. Then, we call `startActivity()` on that `Intent`. Right now, both `help` and `about` do the same thing — we will add some smarts to have them load up different content later in this book.

You will need to add an import for `android.content.Intent` to get this to compile.

If you run this app in a device or emulator, and you choose either the Help or About menu choices... nothing much appears to happen. In reality, what happens is that our `SimpleContentActivity` appeared, but empty, as we have not given it a full UI yet.

In Our Next Episode…

… we will begin using fragments in our tutorial project.

The Tactics of Fragments

Fragments are an optional layer you can put between your activities and your widgets, designed to help you reconfigure your activities to support screens both large (e.g., tablets) and small (e.g., phones).

This chapter will cover basic uses of fragments.

The Six Questions

In the world of journalism, the basics of any news story consist of six questions, the Five Ws and One H. Here, we will apply those six questions to help frame what we are talking about with respect to fragments.

What?

Fragments are not activities, though they can be used by activities.

Fragments are not containers (i.e., subclasses of `ViewGroup`), though typically they create a `ViewGroup`.

Rather, you should think of fragments as being units of UI reuse. You define a fragment, much like you might define an activity, with layouts and lifecycle methods and so on. However, you can then host that fragment in one or several activities, as needed.

Android does not precisely implement UI architectures like Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), etc. To the extent that you wish to shove Android into the MVC architecture, fragments and activities combine to be the controller layer. Fragments serve as a local controller,

focused on their set of widgets, populating them from model data, and handling their events. Activities will serve as more of an orchestration layer, handling cross-fragment communications (e.g., a click in Fragment A needs to cause a change in what is displayed in Fragment B).

Functionally, fragments are Java classes, extending from a base `Fragment` class. As we will see, there are two versions of the `Fragment` class, one native to API Level 11 and one supplied by the Android Support package.

Where??

Since fragments are Java classes, your fragments will reside in one of your application's Java packages. The simplest approach is to put them in the same Java package that you used for your project overall and where your activities reside, though you can refactor your UI logic into other packages if needed.

Who?!?

Typically, you create fragment implementations yourself, then tell Android when to use them. Some third-party Android library projects may ship fragment implementations that you can reuse, if you so choose.

When?!!?

Some developers start adding fragments from close to the outset of application development — that is the approach we will take in the tutorials. And, if you are starting a new application from scratch, defining fragments early on is probably a good idea. That being said, it is entirely possible to "retrofit" an existing Android application to use fragments, though this may be a lot of work. And, it is entirely possible to create Android applications without fragments at all.

Fragments were introduced with Android 3.0 (API Level 11, a.k.a., Honeycomb).

WHY?!?!?

Ah, this is the big question. If we have managed to make it this far through the book without fragments, and we do not necessarily need fragments to create Android applications, what is the point? Why would we bother?

The primary rationale for fragments was to make it easier to support multiple screen sizes.

Android started out supporting phones. Phones may vary in size, from tiny ones with less than 3" diagonal screen size (e.g., Sony Ericsson X10 mini), to monsters that are over 5" (e.g., Samsung Galaxy Note). However, those variations in screen size pale in comparison to the differences between phones and tablets, or phones and TVs.

Some applications will simply expand to fill larger screen sizes. Many games will take this approach, simply providing the user with bigger interactive elements, bigger game boards, etc. Any one of the ever-popular Angry Birds game series, when played on an tablet, gives you bigger birds and bigger pigs, not a phone-sized game area surrounded by ad banners.

However, another design approach is to consider a tablet screen to really be a collection of phone screens, side by side.

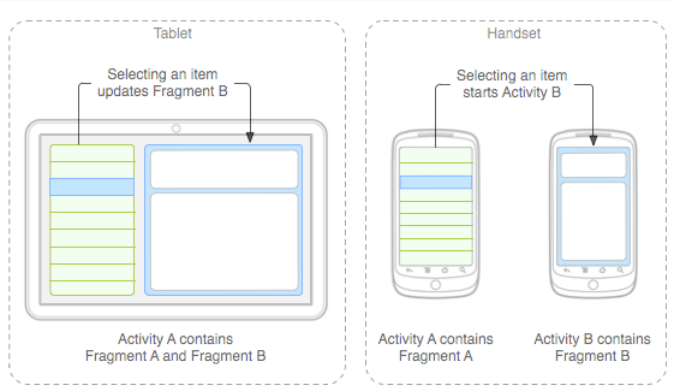


Figure 241: Tablets vs. Handsets (image courtesy of Android Open Source Project)

The user can access all of that functionality at once on a tablet, whereas they would have to flip back and forth between separate screens on a phone.

For applications that can fit this design pattern, fragments allow you to support phones and tablets from one code base. The fragments can be used by individual activities on a phone, or they can be stitched together by a single activity for a tablet.

Details on using fragments to support large screen sizes is a topic for a later chapter in this book. This chapter is focused on the basic mechanics of setting up and using fragments.

OMGOMGOMG, HOW?!?!??

Well, answering that question is what the rest of this chapter is for, plus coverage of more advanced uses of fragments elsewhere in this book.

Where You Get Your Fragments From

Most developers will use the implementation of fragments that has been part of Android since API Level 11. You will use `android.app.Fragment` and have them be hosted by your regular activities.

However, there is a backport of the fragment system available in the Android Support package. This works back to API Level 4, so if your `minSdkVersion` is lower than 11 *and* you want to use fragments, the backport is something that you will wish to consider. You will need to add the `support-v4` JAR to your project (e.g., via `implementation 'com.android.support:support-v4:...'` in your `dependencies` in `build.gradle`, for some value of `...`). You will also need to use `android.support.v4.app.Fragment` instead of `android.app.Fragment`. Also, you will need to host those fragments in an activity inheriting from `android.support.v4.app.FragmentActivity` (as the regular `android.app.Activity` base class does not know about fragments prior to API Level 11).

This book focuses mostly on using the native API Level 11 implementation of fragments, with the occasional example of using the backport where the backport is necessary for one reason or another.

Your First Fragment

In many ways, it is easier to explain fragments by looking at an implementation, more so than trying to discuss them as abstract concepts. So, in this section, we will take a look at the `Fragments/Static` sample project. This is a near-clone of the `Activities/Lifecycle` sample project from the previous chapter. However, we have converted the launcher activity from one that will host widgets directly itself to one that will host a fragment, which in turn manages widgets.

The Fragment Layout

Our fragment is going to manage our UI, so we have a `res/layout/mainfrag.xml` layout file containing our `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/showOther"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="@string/hello"
  android:textSize="20sp"/>
```

(from Fragments/Static/app/src/main/res/layout/mainfrag.xml)

Note, though, that we do not use the `android:onClick` attribute. We will explain why we dropped that attribute from the previous editions of this sample shortly.

The Fragment Class

The project has a `ContentFragment` class that will use this layout and handle the `Button`.

As with activities, there is no constructor on a typical `Fragment` subclass. The primary method you override, though, is not `onCreate()` (though, as we will see later in this chapter, that is possible). Instead, the primary method to override is `onCreateView()`, which is responsible for returning the UI to be displayed for this fragment:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.mainfrag, container, false);

    result.findViewById(R.id.showOther).setOnClickListener(this);

    return(result);
  }
```

(from Fragments/Static/app/src/main/java/com/commonsware/android/sfrag/ContentFragment.java)

We are passed a `LayoutInflater` that we can use for inflating a layout file, the `ViewGroup` that will eventually hold anything we inflate, and the `Bundle` that was passed to the activity's `onCreate()` method. While we are used to framework classes loading our layout resources for us, we can "inflate" a layout resource at any time using a `LayoutInflater`. This process reads in the XML, parses it, walks the element tree, creates Java objects for each of the elements, and stitches the results together into a parent-child relationship.

Here, we inflate `res/layout/mainfrag.xml`, telling Android that its contents will eventually go into the `ViewGroup` but not to add it right away. While there are simpler flavors of the `inflate()` method on `LayoutInflater`, this one is required in case the `ViewGroup` happens to be a `RelativeLayout`, so we can process all of the positioning and sizing rules appropriately.

We also use `findViewById()` to find our `Button` widget and tell it that we, the fragment, are its `OnClickListener`. `ContentFragment` must then implement the `View.OnClickListener` interface to make this work. We do this instead of `android:onClick` to route the `Button` click events to the fragment, not the activity.

Since we implement the `View.OnClickListener` interface, we need the corresponding `onClick()` method implementation:

```
@Override
public void onClick(View v) {
  ((StaticFragmentsDemoActivity)getActivity()).showOther(v);
}
```

(from Fragments/Static/app/src/main/java/com/commonsware/android/sfrag/ContentFragment.java)

Any fragment can call `getActivity()` to find the activity that hosts it. In our case, the only activity that will possibly host this fragment is `StaticFragmentsDemoActivity`, so we can cast the result of `getActivity()` to `StaticFragmentsDemoActivity`, so that we can call methods on our activity. In particular, we are telling the activity to show the other activity, by means of calling the `showOther()` method that we saw in the original `Activities/Lifecycle` sample (and will see again shortly).

That is really all that is needed for this fragment. However, `ContentFragment` also overrides many other fragment lifecycle methods, and we will examine these later in this chapter.

The Activity Layout

Originally, the `res/layout/main.xml` used by the activity was where we had our `Button` widget. Now, the `Button` is handled by the fragment. Instead, our activity layout needs to account for the fragment itself.

In this sample, we are going to use a static fragment. Static fragments are easy to add to your application: just use the `<fragment>` element in a layout file, such as our revised `res/layout/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.sfrag.ContentFragment"/>
```

(from Fragments/Static/app/src/main/res/layout/main.xml)

Here, we are declaring our UI to be completely comprised of one fragment, whose implementation (`com.commonsware.android.sfrag.ContentFragment`) is identified by the `android:name` attribute on the `<fragment>` element. Instead of `android:name`, you can use `class`, though most of the Android documentation has now switched over to `android:name`.

Android Studio users can drag a fragment out of the "Custom" section of the graphical layout editor tool palette, if desired, rather than setting up the `<fragment>` element directly in the XML.

The Activity Class

`StaticFragmentsDemoActivity` — our new launcher activity — looks identical to the previous version, with the exception of the class name:

```
package com.commonsware.android.sfrag;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class StaticFragmentsDemoActivity extends
    LifecycleLoggingActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
    Intent other=new Intent(this, OtherActivity.class);

    other.putExtra(OtherActivity.EXTRA_MESSAGE,
                   getString(R.string.other));
    startActivity(other);
  }
}
```

(from Fragments/Static/app/src/main/java/com/commonsware/android/sfrag/StaticFragmentsDemoActivity.java)

Since the `res/layout/main.xml` file has the `<fragment>` element, the fragment is simply loaded into position in the call to `setContentView()`.

The Fragment Lifecycle Methods

Fragments have lifecycle methods, just like activities do. In fact, they support most of the same lifecycle methods as activities:

- `onCreate()`
- `onStart()` (but not `onRestart()`)
- `onResume()`
- `onPause()`
- `onStop()`
- `onDestroy()`

By and large, the same rules apply for fragments as do for activities with respect to these lifecycle methods (e.g., `onDestroy()` may not be called).

In addition to those and the `onCreateView()` method we examined earlier in this chapter, there are four other lifecycle methods that you can elect to override if you so choose.

`onAttach()` will be called first, even before `onCreate()`, letting you know that your fragment has been attached to an activity. You are passed the `Activity` that will host your fragment.

`onViewCreated()` will be called after `onCreateView()`. This is particularly useful if you are inheriting the `onCreateView()` implementation but need to configure the resulting views, such as with a `ListFragment` and needing to set up an adapter.

`onActivityCreated()` will be called after `onCreate()` and `onCreateView()`, to indicate that the activity's `onCreate()` has completed. If there is something that you need to initialize in your fragment that depends upon the activity's `onCreate()` having completed its work, you can use `onActivityCreated()` for that initialization work.

`onDestroyView()` is called before `onDestroy()`. This is the counterpart to `onCreateView()` where you set up your UI. If there are things that you need to clean up specific to your UI, you might put that logic in `onDestroyView()`.

`onDetach()` is called after `onDestroy()`, to let you know that your fragment has been disassociated from its hosting activity.

onAttach() Versus onAttach()

If you set your project to have a `compileSdkVersion` of 23 or higher, and you attempt to override `onAttach()`, you may get a deprecation warning:

```
@Override
public void onAttach(Activity a) {
  super.onAttach(a);
  Log.d(getClass().getSimpleName(), "onAttach()");
}
```

Figure 242: Android Studio, Showing Deprecated onAttach()

That is because there are *two* versions of `onAttach()` (and `onDetach()`) starting with API Level 23. One takes an `Activity` as a parameter, and the other takes a `Context` as a parameter.

The roles of `onAttach()` and `onDetach()` are the same with either parameter: let you know when the fragment has been attached to or detached from its host. However, now, the host could be anything that extends `Context`, not merely an `Activity`.

On API Level 22 and below, though, only the `Activity` flavor of `onAttach()` and `onDetach()` exists. This leads to a conundrum, as you try to determine exactly how to handle this for your app.

On the whole, if your `minSdkVersion` is below 23, overriding just `onAttach(Activity)` is your best route. It will work on all Android devices that support fragments. Overriding only `onAttach(Context)` will not work, as older devices will ignore it (despite `Activity` being a subclass of `Context`). You could override both methods, but on API Level 23+ devices, both flavors will be called, which may or may not be a good idea for your `Fragment` subclass.

Your First Dynamic Fragment

Static fragments are fairly simple, once you have the `Fragment` implementation: just add the `<fragment>` element to where you want to have the fragment appear in your activity's layout.

That simplicity, though, does come with some costs. We will review some of those limitations in the next chapter.

Those limitations can be overcome by the use of dynamic fragments. Rather than indicating to Android that you wish to use a fragment by means of a `<fragment>` element in a layout, you will use a `FragmentTransaction` to add a fragment at runtime from your Java code.

With that in mind, take a look at the `Fragments/Dynamic` sample project.

This is the same project as the one for static fragments, except this time we will adjust `OtherActivity` to use a dynamic fragment, specifically a `ListFragment`.

The ListFragment Class

`ListFragment` serves the same role for fragments as `ListActivity` does for activities. It wraps up a `ListView` for convenient use. So, to have a more interesting `OtherActivity`, we start with an `OtherFragment` that is a `ListFragment`, designed to show our favorite 25 Latin words as seen in previous examples.

Just as a `ListActivity` does not need to call `setContentView()`, a `ListFragment` does not need to override `onCreateView()`. By default, the entire fragment will be comprised of a single `ListView`. And just as `ListActivity` has a `setListAdapter()` method to associate an `Adapter` with the `ListView`, so too does `ListFragment`:

```
package com.commonsware.android.dfrag;

import android.app.Activity;
import android.app.ListFragment;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ArrayAdapter;

public class OtherFragment extends ListFragment {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };

  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
```

```
    setListAdapter(new ArrayAdapter<String>(getActivity(),
                   android.R.layout.simple_list_item_1, items));
  }
```

(from Fragments/Dynamic/app/src/main/java/com/commonsware/android/dfrag/OtherFragment.java)

We call `setListAdapter()` in `onViewCreated()`, as we know that the `ListView` is now ready for use.

This class also overrides many fragment lifecycle methods, logging their results, akin to our other `Fragment` and `LifecycleLoggingActivity`.

The Activity Class

Now, `OtherActivity` no longer needs to load a layout — we have removed `res/layout/other.xml` from the project entirely. Instead, we will use a `FragmentTransaction` to add our fragment to the UI:

```
package com.commonsware.android.dfrag;

import android.os.Bundle;

public class OtherActivity extends LifecycleLoggingActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new OtherFragment()).commit();
    }
  }
}
```

(from Fragments/Dynamic/app/src/main/java/com/commonsware/android/dfrag/OtherActivity.java)

To work with a `FragmentTransaction`, you need the `FragmentManager`. This object knows about all of the fragments that exist in your activity. If you are using the native API Level 11 edition of fragments, you can get your `FragmentManager` by calling `getFragmentManager()`. If you are using the Android Support package, you need to call `getSupportFragmentManager()` instead.

Given a FragmentManager, you can start a FragmentTransaction by calling beginTransaction(), which returns the FragmentTransaction object. FragmentTransaction operates on the builder pattern, so most methods on FragmentTransaction return the FragmentTransaction itself, so you can chain a series of method calls one after the next.

We call two methods on our FragmentTransaction: add() and commit(). The add() method, as you might guess, indicates that we want to add a fragment to the UI. We supply the actual fragment object, in this case by creating a new OtherFragment. We also need to indicate where in our layout we want this fragment to reside. Had we loaded a layout, we could drop this fragment in any desired container. In our case, since we did not load a layout, we supply android.R.id.content as the ID of the container to hold our fragment's View. Here, android.R.id.content identifies the container into which the results of setContentView() would go — it is a container supplied by Activity itself and serves as the top-most container for our content.

Just calling add() is insufficient. We then need to call commit() to make the transaction actually happen.

You might be wondering why we are trying to find a fragment in our FragmentManager before actually creating the fragment. We do that to help deal with configuration changes, and we will be exploring that further in the next chapter.

Fragments and the Action Bar

Fragments can add items to the action bar by calling setHasOptionsMenu(true) from onCreate() (or any other early lifecycle method). This indicates to the activity that it needs to call onCreateOptionsMenu() and onOptionsItemSelected() on the fragment.

The Fragments/ActionBarNative sample application demonstrates this. This has the same functionality as does the ActionBar/ActionBarDemoNative sample from the chapter on the action bar, just with the activity converted into a dynamic fragment.

In onCreate(), we call setHasOptionsMenu(true), to indicate that we are interested in participating in the action bar:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
```

```
    setRetainInstance(true);
    setHasOptionsMenu(true);
  }
```

(from Fragments/ActionBarNative/app/src/main/java/com/commonsware/android/abf/ActionBarFragment.java)

(we will discuss that `setRetainInstance(true)` call in a later chapter)

That will trigger our fragment's `onCreateOptionsMenu()` and `onOptionsItemSelected()` methods to be called at the appropriate time:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.actions, menu);

    super.onCreateOptionsMenu(menu, inflater);
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
      case R.id.add:
        addWord();

        return(true);

      case R.id.reset:
        initAdapter();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from Fragments/ActionBarNative/app/src/main/java/com/commonsware/android/abf/ActionBarFragment.java)

Here, we initialize our action bar from the `R.menu.actions` menu XML resource, along with the logic to respond to the `add` and `reset` action bar items.

Our activity does not need to do anything special to allow the fragment to contribute to the action bar — it just sets up the dynamic fragment:

```
package com.commonsware.android.abf;

import android.app.Activity;
import android.os.Bundle;
```

```
public class ActionBarFragmentActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new ActionBarFragment()).commit();
    }
  }
}
```

(from Fragments/ActionBarNative/app/src/main/java/com/commonsware/android/abf/ActionBarFragmentActivity.java)

Fragments Within Fragments: Just Say "Maybe"

Historically, one major limitation with fragments is that they could not contain other fragments. In most cases, this does not pose a major problem. However, there will be times when you might trip over this limitation, such as when using a `ViewPager`, as will be described in a later chapter.

Android 4.2 — and the Android Support package – added support for nested fragments. Whereas an activity works with fragments via a `FragmentManager` obtained via `getFragmentManager()` or `getSupportFragmentManager()`, fragments can work with nested fragments via a call to `getChildFragmentManager()`.

However, Android 3.0 through 4.1 have a version of fragments that does not have `getChildFragmentManager()`. Hence, you have two options:

1. Use the Android Support package's backport of fragments, until such time as you can drop support for Android 4.1 and earlier, or
2. Do not use nested fragments for the time being.

We will see how `getChildFragmentManager()` works in the chapter on `ViewPager`.

Fragments and Multiple Activities

A fragment should handle functionality purely within the fragment itself. Anything outside the fragment should be the responsibility of the calling activity. For example, if the user taps on an item in a `ListFragment`, and the effects of that event might go

beyond what is inside the `ListFragment` itself, the `ListFragment` should forward the event to the hosting activity, so it can perhaps perform additional steps (e.g., launch an activity, update another fragment hosted by the activity).

As we will see in a later chapter, it is entirely possible — perhaps even likely — that some of our fragments will be hosted by multiple different activities. For example, we might have a fragment that is hosted in one case by an activity designed for larger screens (e.g., tablets) and in another case by an activity designed for smaller screens (e.g., phones).

In these cases, the fragment does not know at compile time which activity class will be hosting it at runtime. For those cases, you have three major options:

1. Have the activities implement a common interface, and have the fragment cast the result of calling `getActivity()` to that interface, so it can call methods on the hosting activity without knowing its exact implementation.
2. Have the activities supply a listener object, with a common interface, to the fragment via a setter, and have the fragment use that listener for raising events and so on.
3. Use an event bus, as we will explore later in this book.

We will see much more on this subject when we get into large-screen strategies in a later chapter.

Tutorial #9 - Starting Our Fragments

Much of the content of a digital book to be viewed in EmPubLite will be in the form of HTML and related assets (CSS, images, etc.). Hence, we will eventually need to render our content in a `WebView` widget, for best results with semi-arbitrary HTML content.

To do this, we will set up fragments for the bits of content:

- each chapter (or, in our case, HTML file containing chapters)
- other material, like our "help" and "about" pages

Right now, we will focus on just setting up some of the basic classes for these fragments — we will load them up with content and display them over the next few tutorials.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Create a SimpleContentFragment

Android has a `WebViewFragment` for the native API Level 11+ implementation of fragments, designed to show some Web content in a `WebView`. In this step, we will create a subclass of `WebViewFragment` that adds in a bit of EmPubLite-specific business logic.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > "Java Class" from the context menu. That will bring up a new-class dialog:

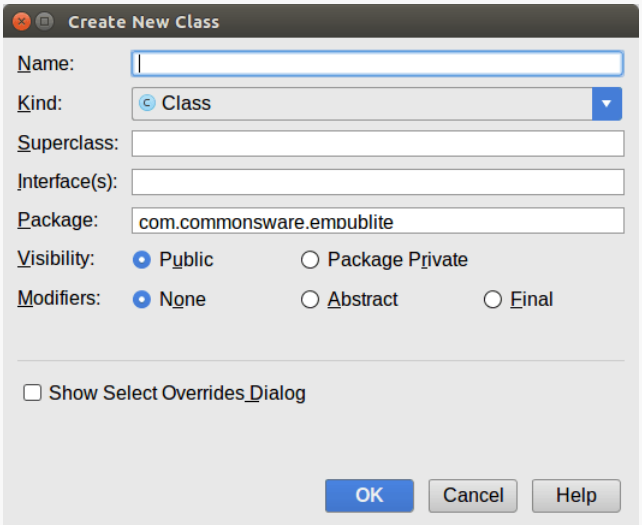


Figure 243: Android Studio New Class Dialog

Fill in `SimpleContentFragment` for the name and `android.app.Fragment` for the superclass. Then, click OK to create this class.

Then, replace the contents of the fragment class with the following code:

```
package com.commonsware.empublite;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.webkit.WebViewFragment;

public class SimpleContentFragment extends WebViewFragment {
  private static final String KEY_FILE="file";

  static SimpleContentFragment newInstance(String file) {
    SimpleContentFragment f=new SimpleContentFragment();

    Bundle args=new Bundle();

    args.putString(KEY_FILE, file);
```

```
    f.setArguments(args);

    return(f);
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
  }

  @SuppressLint("SetJavaScriptEnabled")
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
      super.onCreateView(inflater, container, savedInstanceState);

    getWebView().getSettings().setJavaScriptEnabled(true);
    getWebView().getSettings().setSupportZoom(true);
    getWebView().getSettings().setBuiltInZoomControls(true);
    getWebView().loadUrl(getPage());

    return(result);
  }

  private String getPage() {
    return(getArguments().getString(KEY_FILE));
  }
}
```

(from EmPubLite-AndroidStudio/T9-Fragments/EmPubLite/app/src/main/java/com/commonsware/empublite/SimpleContentFragment.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #2: Examining SimpleContentFragment

`SimpleContentFragment` is simple, with a total of four methods:

- `onCreate()`, where we call `setRetainInstance(true)` — the utility of this will be examined in greater detail in an upcoming chapter.
- `onCreateView()`, where we chain to the superclass (to have it create the `WebView`), then configure it to accept JavaScript and support zoom

operations. We then have it load some content, retrieved in the form of a URL from a private `getPage()` method. Finally, we return what the superclass returned from `onCreateView()` — effectively, we are simply splicing in our own configuration logic.

- a `newInstance()` static factory method. This method creates an instance of `SimpleContentFragment`, takes a passed-in `String` (pointing to the file to load), puts it in a `Bundle` identified as `KEY_FILE`, hands the `Bundle` to the fragment as its arguments, and returns the newly-created `SimpleContentFragment`.
- `getPage()`, where it returns a value out of the "arguments" `Bundle` supplied to the fragment — specifically the string identified as `KEY_FILE`.

This means that anyone wanting to use `SimpleContentFragment` should use the factory method, to provide the path to the content to load. We will see why we implemented `SimpleContentFragment` this way in the next chapter.

In Our Next Episode…

… we will set up horizontal swiping of book chapters in our tutorial project.

Swiping with ViewPager

Android, over the years, has put increasing emphasis on UI design and having a fluid and consistent user experience (UX). While some mobile operating systems take "the stick" approach to UX (forcing you to abide by certain patterns or be forbidden to distribute your app), Android takes "the carrot" approach, offering widgets and containers that embody particular patterns that they espouse. The action bar, for example, grew out of this and is now the backbone of many Android activities.

Another example is the `ViewPager`, which allows the user to swipe horizontally to move between different portions of your content. However, `ViewPager` is not distributed as part of the firmware, but rather via the Android Support package. Hence, even though `ViewPager` is a relatively new widget, you can use it on Android 1.6 and up.

This chapter will focus on where you should apply a `ViewPager` and how to set one up.

Pieces of a Pager

`AdapterView` classes, like `ListView`, work with `Adapter` objects, like `ArrayAdapter`. `ViewPager`, however, is not an `AdapterView`, despite adopting many of the patterns from `AdapterView`. `ViewPager`, therefore, does not work with an `Adapter`, but instead with a `PagerAdapter`, which has a slightly different API.

Android ships two `PagerAdapter` implementations in the Android Support package: `FragmentPagerAdapter` and `FragmentStatePagerAdapter`. The former is good for small numbers of fragments, where holding them all in memory at once will work. `FragmentStatePagerAdapter` is for cases where holding all possible fragments to be

viewed in the ViewPager would be too much, where Android will discard fragments as needed and hold onto the (presumably smaller) states of those fragments instead.

Paging Fragments

The simplest way to use a ViewPager is to have it page fragments in and out of the screen based on user swipes.

To see this in action, this section will examine the ViewPager/Fragments sample project.

The project has a dependency on the Android Support package, in order to be able to use ViewPager. In Android Studio, this is a implementation statement in the dependencies closure of build.gradle:

```
dependencies {
  implementation "com.android.support:support-fragment:27.0.2"
  implementation "com.android.support:support-core-ui:27.0.2"
```

(from ViewPager/Fragments/app/build.gradle)

The Activity Layout

The layout used by the activity just contains the ViewPager. Note that since ViewPager is not in the android.widget package, we need to fully-qualify the class name in the element:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
</android.support.v4.view.ViewPager>
```

(from ViewPager/Fragments/app/src/main/res/layout/main.xml)

Note that ViewPager is not available for drag-and-drop in the IDE graphical designers, probably because it comes from the Android Support package and therefore is not available to all projects.

The Activity

As you see, the ViewPagerFragmentDemoActivity itself is blissfully small:

```
package com.commonsware.android.pager;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.support.v4.view.ViewPager;

public class ViewPagerFragmentDemoActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=findViewById(R.id.pager);

    pager.setAdapter(new SampleAdapter(getSupportFragmentManager()));
  }
}
```

(from ViewPager/Fragments/app/src/main/java/com/commonsware/android/pager/ViewPagerFragmentDemoActivity.java)

All we do is load the layout, retrieve the `ViewPager` via `findViewById()`, and provide a `SampleAdapter` to the `ViewPager` via `setAdapter()`.

The PagerAdapter

Our `SampleAdapter` inherits from `FragmentPagerAdapter` and implements two required callback methods:

- `getCount()`, to indicate how many pages will be in the `ViewPager`, and
- `getItem()`, which returns a `Fragment` for a particular position within the `ViewPager` (akin to `getView()` in a classic `Adapter`)

```
package com.commonsware.android.pager;

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  public SampleAdapter(FragmentManager mgr) {
    super(mgr);
  }

  @Override
  public int getCount() {
    return(10);
```

```
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }
}
```

(from ViewPager/Fragments/app/src/main/java/com/commonsware/android/pager/SampleAdapter.java)

Here, we say that there will be 10 pages total, each of which will be an instance of an `EditorFragment`. In this case, rather than use the constructor for `EditorFragment`, we are using a `newInstance()` factory method. The rationale for that will be explained in the next section.

The Fragment

`EditorFragment` will host a full-screen `EditText` widget, for the user to enter in a chunk of prose, as is defined in the `res/layout/editor.xml` resource:

```
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/editor"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:inputType="textMultiLine"
  android:gravity="left|top"
  />
```

(from ViewPager/Fragments/app/src/main/res/layout/editor.xml)

We want to pass the position number of the fragment within the `ViewPager`, simply to customize the hint displayed in the `EditText` before the user types in anything. With normal Java objects, you might pass this in via the constructor, but it is not a good idea to implement a constructor on a `Fragment`. Instead, the recipe is to create a static factory method (typically named `newInstance()`) that will create the `Fragment` and provide the parameters to it by updating the fragment's "arguments" (a `Bundle`):

```
  static EditorFragment newInstance(int position) {
    EditorFragment frag=new EditorFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);
```

```
    return(frag);
  }
```

(from ViewPager/Fragments/app/src/main/java/com/commonsware/android/pager/EditorFragment.java)

You might be wondering why we are bothering with this `Bundle`, instead of just using a regular data member. The arguments `Bundle` is part of our "saved instance state", for dealing with things like screen rotations — a concept we will get into later in the book. For the moment, take it on faith that this is a good idea.

In `onCreateView()` we inflate our `R.layout.editor` resource, get the `EditText` from it, get our position from our arguments, format a hint containing the position (using a string resource), and setting the hint on the `EditText`:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);
  EditText editor=result.findViewById(R.id.editor);
  int position=getArguments().getInt(KEY_POSITION, -1);

  editor.setHint(String.format(getString(R.string.hint), position + 1));

  return(result);
}
```

(from ViewPager/Fragments/app/src/main/java/com/commonsware/android/pager/EditorFragment.java)

The Result

When initially launched, the application shows the first fragment:

Figure 244: ViewPager on Android 4.3, Showing First Editor

However, you can horizontally swipe to get to the next fragment:

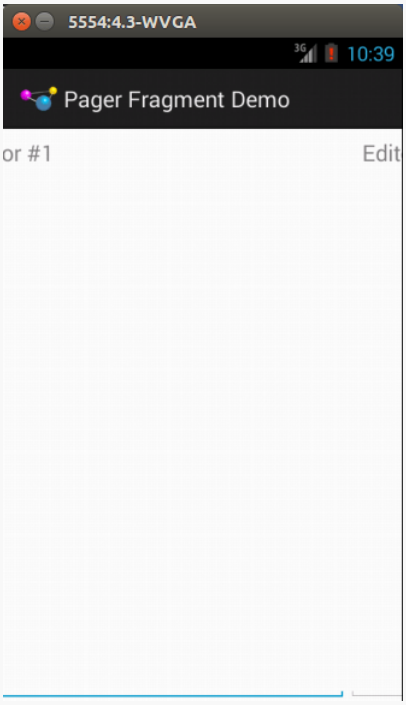


Figure 245: A ViewPager in Use on Android 4.0.3

Swiping works in both directions, so long as there is another page in your desired direction.

Paging Other Stuff

You do not have to use fragments inside a `ViewPager`. A regular `PagerAdapter` actually hands `View` objects to the `ViewPager`. The supplied fragment-based `PagerAdapter` implementations get the `View` from a fragment and use that, but you are welcome to create your own `PagerAdapter` that avoids fragments.

Hence, if you want `ViewPager` to page things other than fragments, the solution is to not use `FragmentPagerAdapter` or `FragmentStatePagerAdapter`, but instead create your own implementation of the `PagerAdapter` interface, one that avoids the use of fragments.

We will see an example of this in a later chapter, where we also examine how to have more than one page of the `ViewPager` be visible at a time.

Indicators

By itself, there is no visual indicator of where the user is within the set of pages contained in the `ViewPager`. In many instances, this will be perfectly fine, as the pages themselves will contain cues as to position. However, even in those cases, it may not be completely obvious to the user how many pages there are, which directions for swiping are active, etc.

Hence, you may wish to attach some other widget to the `ViewPager` that can help clue the user into where they are within "page space".

PagerTitleStrip and PagerTabStrip

The primary built-in indicator options available to use are `PagerTitleStrip` and `PagerTabStrip`. As the name suggests, `PagerTitleStrip` is a strip that shows titles of your pages. `PagerTabStrip` is much the same, but the titles are formatted somewhat like tabs, and they are clickable (switching you to the clicked-upon page), whereas `PagerTitleStrip` is non-interactive.

To use either of these, you first must add it to your layout, inside your `ViewPager`, as shown in the `res/layout/main.xml` resource of the `ViewPager/Indicator` sample project, a clone of the `ViewPager/Fragments` project that adds a `PagerTabStrip` to our UI:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>

</android.support.v4.view.ViewPager>
```

(from ViewPager/Indicator/app/src/main/res/layout/main.xml)

Here, we set the `android:layout_gravity` of the `PagerTabStrip` to `top`, so it appears above the pages. You could similarly set it to `bottom` to have it appear below the pages.

Our `SampleAdapter` needs another method: `getPageTitle()`, which will return the title to display in the `PagerTabStrip` for a given position:

```
package com.commonsware.android.pager2;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  Context ctxt=null;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(10);
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }

  @Override
  public String getPageTitle(int position) {
    return(EditorFragment.getTitle(ctxt, position));
  }
}
```

(from ViewPager/Indicator/app/src/main/java/com/commonsware/android/pager2/SampleAdapter.java)

Here, we call a static `getTitle()` method on `EditorFragment`. That is a refactored bit of code from our former `onCreateView()` method, where we create the string for the hint — we will use the hint text as our page title:

```
package com.commonsware.android.pager2;

import android.app.Fragment;
import android.content.Context;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
```

```
public class EditorFragment extends Fragment {
  private static final String KEY_POSITION="position";

  static EditorFragment newInstance(int position) {
    EditorFragment frag=new EditorFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);

    return(frag);
  }

  static String getTitle(Context ctxt, int position) {
    return(String.format(ctxt.getString(R.string.hint), position + 1));
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);
    EditText editor=(EditText)result.findViewById(R.id.editor);
    int position=getArguments().getInt(KEY_POSITION, -1);

    editor.setHint(getTitle(getActivity(), position));

    return(result);
  }
}
```

(from ViewPager/Indicator/app/src/main/java/com/commonsware/android/pager2/EditorFragment.java)

Figure 246: ViewPager and PagerTabStrip on Android 4.3, Showing Second Page

Other Indicator Options

There are many other options for tab-style indicators with a `ViewPager`.

`TabLayout` is the only other one officially supported by Google. It comes from the Design Support library, which in turn requires `appcompat-v7`.

Most of the options are open source libraries. The chapter on the Design Support library demonstrates one such third-party library, The Android Arsenal's roster of `ViewPager` add-ons mostly consists of indicators, both those formatted like tabs and those that use other approaches.

Revisiting the Containers Sampler

Earlier in the book, we looked at many different layout resources from the `Containers/Sampler` sample project. These were to illustrate how different layout structures work. The whole UI is wrapped up in a `ViewPager` with a `PagerTabStrip`. However, we set up the `FragmentPagerAdapter` and the fragments based on a roster of available sample layouts, driven by resources.

The Layout

The layout that is used for the `ViewPager` and `PagerTabStrip` is largely the same as what we saw earlier in this chapter:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>

</android.support.v4.view.ViewPager>
```

(from Containers/Sampler/app/src/main/res/layout/main.xml)

So, we have a `ViewPager` named `pager`, with a `PagerTabStrip` anchored at the top via `android:layout_gravity="top"`.

The Data

The data about what layouts to show as tabs is contained in a pair of array resources. Array resources, as the name suggests, are resources that hold onto a collection of items. The convention is that they go in `res/values/arrays.xml`, though the actual filename is not required to be `arrays.xml`.

The sample app has a `res/values/arrays.xml` file, containing two array resources: a `<string-array>` named `titles` and a generic `<array>` named `layouts`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="titles">
    <item>No Container</item>
    <item>Bottom-then-Top: LinearLayout</item>
    <item>Bottom-then-Top: RelativeLayout</item>
    <item>Bottom-then-Top: ConstraintLayout</item>
    <item>Stacked Percent: LinearLayout</item>
    <item>Stacked Percent: ConstraintLayout</item>
    <item>URL Dialog: LinearLayout</item>
    <item>URL Dialog: RelativeLayout</item>
    <item>URL Dialog: TableLayout</item>
    <item>URL Dialog: ConstraintLayout</item>
    <item>Form: TableLayout</item>
```

```
    <item>Form: LinearLayout</item>
    <item>Overlap: RelativeLayout</item>
    <item>Center: RelativeLayout</item>
    <item>Center: ConstraintLayout</item>
    <item>Bias: ConstraintLayout</item>
    <item>Bias: LinearLayout</item>
    <item>Bias: ConstraintLayout Peers</item>
    <item>Aspect: ConstraintLayout</item>
    <item>Center Align: LinearLayout</item>
    <item>Center Align: ConstraintLayout</item>
    <item>Chains: ConstraintLayout</item>
  </string-array>
  <array name="layouts">
    <item>@layout/no_container</item>
    <item>@layout/bottom_then_top_ll</item>
    <item>@layout/bottom_then_top_rl</item>
    <item>@layout/bottom_then_top_cl</item>
    <item>@layout/stacked_percent_ll</item>
    <item>@layout/stacked_percent_cl</item>
    <item>@layout/url_dialog_ll</item>
    <item>@layout/url_dialog_rl</item>
    <item>@layout/url_dialog_tl</item>
    <item>@layout/url_dialog_cl</item>
    <item>@layout/form_tl</item>
    <item>@layout/form_ll</item>
    <item>@layout/overlap_rl</item>
    <item>@layout/center_rl</item>
    <item>@layout/center_cl</item>
    <item>@layout/bias_cl</item>
    <item>@layout/bias_ll</item>
    <item>@layout/bias_peers_cl</item>
    <item>@layout/aspect_cl</item>
    <item>@layout/center_align_ll</item>
    <item>@layout/center_align_cl</item>
    <item>@layout/chains_cl</item>
  </array>
</resources>
```

(from Containers/Sampler/app/src/main/res/values/arrays.xml)

As the name suggests, a `<string-array>` holds strings. Here, it holds literal strings. Alternatively, those could be references to string resources, using the same `@string/...` naming convention that we have seen elsewhere in the book.

The `layouts` array holds references to the corresponding layout resources, using `@layout/...` syntax to identify those resources.

These two arrays are set up in the same order, so the first title is used for the first layout, the second title is used for the second layout, and so on.

The Fragment

The fragment for the pages in the `ViewPager` is a `static` nested class within the `MainActivity`, named `LayoutFragment`. This is atypical; usually a fragment will be in its own Java class. That is because usually a fragment has a lot of code associated with it. In this case, `LayoutFragment` is rather short:

```
public static class LayoutFragment extends Fragment {
  private static final String ARG_LAYOUT="layout";

  static LayoutFragment newInstance(int layoutId) {
    LayoutFragment result=new LayoutFragment();
    Bundle args=new Bundle();

    args.putInt(ARG_LAYOUT, layoutId);
    result.setArguments(args);

    return(result);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    return(inflater.inflate(getArguments().getInt(ARG_LAYOUT),
      container, false));
  }
}
```

(from Containers/Sampler/app/src/main/java/com/commonsware/android/containers/sampler/MainActivity.java)

When we create an instance of `LayoutFragment` through the `newInstance()` factory method, we pass in a layout resource ID. Through the arguments `Bundle`, that becomes available to our `onCreateView()` method, which simply inflates that layout and returns it.

The fragment is so short because we are not doing anything with the widgets inside of the inflated layout, such as filling them with data or setting up event listeners.

The Activity

The `MainActivity` that contains `LayoutFragment` has only one method of its own: `onCreate()`, which inflates the `res/layout/main.xml` resource shown above, finds the `ViewPager`, and sets its adapter to be a `SampleAdapter`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  ViewPager pager=(ViewPager)findViewById(R.id.pager);

  pager.setAdapter(new SampleAdapter(getFragmentManager()));
}
```

(from Containers/Sampler/app/src/main/java/com/commonsware/android/containers/sampler/MainActivity.java)

In other words, this `onCreate()` is pretty much the same as the others seen in this chapter. Where the fun lies is in this sample app's edition of `SampleAdapter`.

The PagerAdapter

`SampleAdapter` uses the contents of those two array resources to determine how many pages there are, what to show in the tabs, and what layout resource ID to pass to the `LayoutFragment`:

```
private class SampleAdapter extends FragmentPagerAdapter {
  private int[] layouts;
  private String[] titles;

  SampleAdapter(FragmentManager mgr) {
    super(mgr);
    layouts=getLayoutsArray(R.array.layouts);
    titles=getResources().getStringArray(R.array.titles);
  }

  @Override
  public int getCount() {
    return(titles.length);
  }

  @Override
  public Fragment getItem(int position) {
    return(LayoutFragment.newInstance(layouts[position]));
```

```
  }

  @Override
  public CharSequence getPageTitle(int position) {
    return(titles[position]);
  }

  int[] getLayoutsArray(int arrayResourceId) {
    TypedArray typedArray=
      getResources().obtainTypedArray(arrayResourceId);
    int[] result=new int[typedArray.length()];

    for (int i=0;i<typedArray.length();i++) {
      result[i]=typedArray.getResourceId(i, -1);
    }

    return(result);
  }
}
```

(from Containers/Sampler/app/src/main/java/com/commonsware/android/containers/sampler/MainActivity.java)

`SampleAdapter` holds onto a `String` array of the titles. This is populated by `getResources().getStringArray()` in the `SampleAdapter` constructor, which reads in the `titles` `<string-array>` resource. Those strings are then used for the `getPageTitle()` method, and the length of the `strings` array is used for `getCount()`.

Getting the equivalent array of the `int` values for the layout resource IDs is more cumbersome. We could try calling `getIntArray()` on the `Resources` object returned by `getResources()`. However, that only works for `<int-array>` resources. We cannot use one of those in `res/values/arrays.xml` because Android considers `@layout/...` to be a resource ID, not just a simple integer.

To work with a generic `<array>` resource, we need to call `obtainTypedArray()` on the `Resources` object, which we do in the `getLayoutsArray()` helper method. This returns a `TypedArray`, which effectively is an array of arbitrary resource types. We know that our array should be all layout resource IDs, so we allocate an `int` array to be our result (based upon the `length()` of the `TypedArray`), then iterate over all of the entries and retrieve the resource ID for that array position (using `getResourceId()`). In the end, `result` has an array of resource IDs, and those get used by `getItem()` to supply `LayoutFragment` with the appropriate layout resource ID for this page's position.

Visit the Trails!

There is a chapter on advanced `ViewPager` techniques that may interest you!

Tutorial #10 - Rigging Up a ViewPager

A `ViewPager` is a fairly slick way to present a digital book. You can have individual portions of the book be accessed by horizontal swiping, with the prose within a portion accessed by scrolling vertically. While not offering "page-at-a-time" models used by some book reader software, it is **much** simpler to set up.

So, that's the approach we will use with EmPubLite. Which means, among other things, that we need to add a `ViewPager` to the app.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Add a ViewPager to the Layout

Right now, the `main` layout used by `EmPubLiteActivity` just has a `TextView`. We need to change that to have our `ViewPager`.

Since `ViewPager` is not available for drag-and-drop through the IDE graphical layout editors, even IDE users are going to have to dive into the layout XML this time.

Open up `res/layout/main.xml` and switch to the Text sub-tab to see the raw XML. Replace its contents with:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
```

```
  android:orientation="vertical">

  <io.karim.MaterialTabs
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="48dp"
    app:mtIndicatorColor="@color/colorAccent"
    app:mtSameWeightTabs="true"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>
</LinearLayout>
```

(from EmPubLite-AndroidStudio/T10-ViewPager/EmPubLite/app/src/main/res/layout/main.xml)

This adds our `ViewPager`, underneath a `MaterialTabs`, inside a `vertical` `LinearLayout`. `MaterialTabs` is an implementation of tabs for a `ViewPager`, one that we will explore in greater detail later in the book.

Step #2: Creating a ContentsAdapter

A `ViewPager` needs a `PagerAdapter` to populate its content, much like a `ListView` needs a `ListAdapter`. We cannot completely construct a `PagerAdapter` yet, as we still need to learn how to load up our book content from files. But, we can get part-way towards having a useful `PagerAdapter` now.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `ContentsAdapter` as the name and click OK to create the empty class.

Then, replace the generated `ContentsAdapter.java` file with the following content:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.support.v13.app.FragmentStatePagerAdapter;

public class ContentsAdapter extends FragmentStatePagerAdapter {
  public ContentsAdapter(Activity ctxt) {
    super(ctxt.getFragmentManager());
  }
```

```
  @Override
  public Fragment getItem(int arg0) {
// TODO Auto-generated method stub
    return null;
  }

  @Override
  public int getCount() {
// TODO Auto-generated method stub
    return 0;
  }
}
```

(from EmPubLite-AndroidStudio/T10-ViewPager/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #3: Setting Up the ViewPager

Now, we need to add some code to retrieve the `ViewPager`, populate it with the `ContentsAdapter`, and do something useful with those tabs.

First, add two fields to `EmPubLiteActivity`:

```
  private ViewPager pager;
  private ContentsAdapter adapter;
```

(from EmPubLite-AndroidStudio/T10-ViewPager/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

This will require adding an `import` for `android.support.v4.view.ViewPager`.

Then, add a few more lines to the bottom of `onCreate()` of `EmPubLiteActivity`:

```
    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new ContentsAdapter(this);
    pager.setAdapter(adapter);

    MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);
```

(from EmPubLite-AndroidStudio/T10-ViewPager/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

This will require an `import` statement for `io.karim.MaterialTabs`.

What we are doing is:

- Retrieving our `ViewPager`, holding onto it in its field,
- Creating an instance of the do-nothing `ContentsAdapter`,
- Associating the `ContentsAdapter` with the `ViewPager`,
- Retrieving the `MaterialTabs`, and
- Attaching the tabs to the `ViewPager`

At this point, your `EmPubLiteActivity` should look something like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.view.ViewPager;
import android.view.Menu;
import android.view.MenuItem;
import io.karim.MaterialTabs;

public class EmPubLiteActivity extends Activity {
  private ViewPager pager;
  private ContentsAdapter adapter;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new ContentsAdapter(this);
    pager.setAdapter(adapter);

    MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);

    return(super.onCreateOptionsMenu(menu));
  }
```

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.about:
        Intent i=new Intent(this, SimpleContentActivity.class);
        startActivity(i);

        return(true);

      case R.id.help:
        i=new Intent(this, SimpleContentActivity.class);
        startActivity(i);

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
}
```

(from EmPubLite-AndroidStudio/T10-ViewPager/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

The net effect, if you run this modified version of the app, is that we have a big blank area, taken up by our empty `ViewPager`:

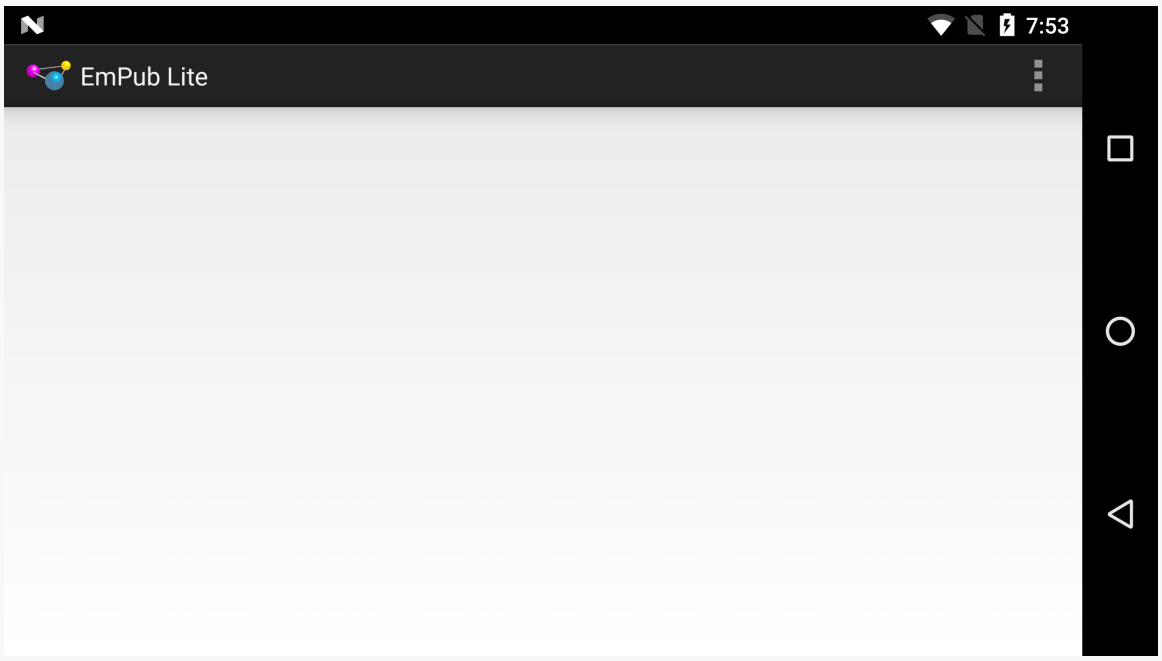


Figure 247: EmPubLite, With Empty ViewPager

The `ViewPager` is empty simply because our `ContentsAdapter` returned `0` from `getCount()`, indicating that there are no pages to be displayed.

In Our Next Episode…

… we will finish our "help" and "about" screens in our tutorial project.

Resource Sets and Configurations

Devices sometimes change while users are using them, in ways that our application will care about:

- The user might rotate the screen from portrait to landscape, or vice versa
- The user might put the device in a car or desk dock, or remove it from such a dock
- The user might put the device in a "netbook dock" that adds a full QWERTY keyboard, or remove it from such a dock
- The user might switch to a different language via the Settings application, returning to our running application afterwards
- And so on

In all of these cases, it is likely that we will want to change what resources we use. For example, our layout for a portrait screen may be too tall to use in landscape mode, so we would want to substitute in some other layout.

This chapter will explore how to provide alternative resources for these different scenarios — called "configuration changes" — and will explain what happens to our activities when the user changes the configuration while we are in the foreground.

What's a Configuration? And How Do They Change?

Different pieces of Android hardware can have different capabilities, such as:

- Different screen sizes
- Different screen densities (dots per inch)

- Different number and capabilities of cameras
- Different mix of radios (GSM? CDMA? GPS? Bluetooth? WiFi? NFC? something else?)
- And so on

Some of these, in the eyes of the core Android team, might drive the selection of resources, like layouts or drawables. Different screen sizes might drive the choice of layout. Different screen densities might drive the choice of drawable (using a higher-resolution image on a higher-density device). These are considered part of the device's "configuration".

Other differences — ones that do not drive the selection of resources — are not part of the device's configuration but merely are "features" that some devices have and other devices do not. For example, cameras and Bluetooth and WiFi are features.

Some parts of a configuration will only vary based on different devices. A screen will not change density on the fly, for example. But some parts of a configuration can be changed during operation of the device, such as orientation (portrait vs. landscape) or language. When a configuration switches to something else, that is a "configuration change", and Android provides special support for such events to help developers adjust their applications to match the new configuration.

Configurations and Resource Sets

One set of resources may not fit all situations where your application may be used. One obvious area comes with string resources and dealing with internationalization (I18N) and localization (L10N). Putting strings all in one language works fine — probably at least for the developer — but only covers one language.

That is not the only scenario where resources might need to differ, though. Here are others:

1. *Screen orientation*: is the screen in a portrait orientation? Landscape?
2. *Screen size*: is this something sized like a phone? A tablet? A television?
3. *Screen density*: how many dots per inch does the screen have? Will we need a higher-resolution edition of our icon so it does not appear too small?
4. *Keyboard*: what keyboard does the user have (QWERTY, numeric, neither), either now or as an option?
5. *Other input*: does the device have some other form of input, like a directional pad or click-wheel?

The way Android currently handles this is by having multiple resource directories, with the criteria for each embedded in their names.

Suppose, for example, you want to support strings in both English and Spanish. Normally, for a single-language setup, you would put your strings in a file named `res/values/strings.xml`. To support both English and Spanish, you could create two folders, `res/values-en/` and `res/values-es/`, where the value after the hyphen is the ISO 639-1 two-letter code for the language you want. Your English-language strings would go in `res/values-en/strings.xml` and the Spanish ones in `res/values-es/strings.xml`. Android will choose the proper file based on the user's device settings. Note that Android 5.0 added support for BCP 47 three-letter language and locale values.

However, the **better** approach is for you to consider some language to be your default, and put those strings in `res/values/strings.xml`. Then, create other resource directories for your translations (e.g., `res/values-es/strings.xml` for Spanish). Android will try to match a specific language set of resources; failing that, it will fall back to the default of `res/values/strings.xml`. This way, if your app winds up on a device with a language that you do not expect, you at least serve up strings in your chosen default language. Otherwise, if there is no such default, you will wind up with a `ResourceNotFoundException`, and your application will crash.

This, therefore, is the bedrock resource set strategy: have a complete set of resources in the default directory (e.g., `res/layout/`), and override those resources in other resource sets tied to specific configurations as needed (e.g., `res/layout-land/`).

Note that Android Studio has a translations editor to help you manage your string resources for your default language and whatever translations you are going to include in your app.

Screen Size and Orientation

Perhaps the most important resource set qualifiers that we have not yet seen are the ones related to screen size and orientation. Here, "orientation" refers to how the device is being held: portrait or landscape.

Orientation is fairly easy, as you can just use `-port` or `-land` as resource set qualifiers to restrict resources in a directory to a specific orientation. The convention is to put landscape resources in a `-land` directory (e.g., `res/layout-land/`) and to put

portrait resource in the default directory (e.g., `res/layout/`). However, this is merely a convention, and you are welcome to use `-port` if you prefer.

Screen size is a bit more complicated, simply because the available approaches have changed over the years.

The Original: Android-Defined Buckets

Way back in the beginning, with Android 1.0, all screen sizes were created equal... mostly because there was only one screen size, and that mostly because there was only one device.

Android 1.5, however, introduced three screen sizes and associated resource set qualifiers, with a fourth (`-xlarge`) added later:

- `-small` for screens at or under 3" in diagonal size
- `-normal` for screens between 3" and 5" in diagonal size
- `-large` for screens between 5" and 10" in diagonal size
- `-xlarge` for screens at or over 10" in diagonal size

As we will see, these resource set qualifiers establish lower bounds for when a directory's worth of resources will be used. So a `res/layout-normal/` directory will not be used for `-small` screens but would be used for `-normal`, `-large`, and `-xlarge` screens.

The Modern: Developer-Defined Buckets

The problem with the classic size buckets is that they were fairly inflexible. What if you think that so-called "phablets", like the Samsung Galaxy Note series, should have layouts more like phones, while larger tablets, such as the 8.9" Kindle Fire HD, should have layouts more like 10" tablets? That was not possible given the fixed buckets.

Android 3.2 gave us more control. We can have our own buckets for screen size, using the somewhat-confusing `-swNNNdp` resource set qualifier. Here, the `NNN` is replaced by you with a value, measured in `dp`, for the *smallest width* of the screen. "Smallest width" basically means the width of the screen when the device is held in portrait mode. Hence, rather than measuring based on diagonal screen size, as with the classic buckets, your custom buckets are based on the linear screen size of the shortest screen side.

For example, suppose that you wish to consider a dividing line between resources to be at the 7" point — 7" and smaller devices would get one set of layouts, while larger devices would get a different set of layouts. 7" tablets usually have a smallest width of around 3.5" to 3.75", given common aspect ratios. Since 1 `dp` is 1/160th of an inch, those smallest widths equate to 560-600 `dp`. Hence, you might set up a `-sw600dp` resource set for your larger layouts, and put the smaller layouts in a default resource set.

Mashups: Width and Height Buckets

Using `-swNNNdp` does not address orientation, as the smallest width is the same regardless of whether the device is held in portrait or landscape. Hence, you would need to add `-swNNNdp-land` as a resource set for landscape resources for your chosen dividing line.

An alternative is to use `-wNNNdp` or `-hNNNdp`. These resource set qualifiers work much like `-swNNNdp`, particularly in terms of what `NNN` means. However, whereas `-swNNNdp` refers to the smallest width, `-wNNNdp` refers the *current* width, and `-hNNNdp` refers to the *current* height. Hence, these change with orientation changes.

About That API Level

`-swNNNdp`, `-wNNNdp`, and `-hNNNdp` were added in API Level 13. Hence, older devices will ignore any resource sets with those qualifiers.

In principle, this might seem like a big problem, for those developers still supporting older devices.

In practice, it is less of an issue than you might expect, simply because the *vast* majority of those older devices were phones, not tablets. The only Android 2.x tablets that sold in any significant quantity were three 7" models:

- the original Kindle Fire
- the original Barnes & Noble NOOK series
- the original Samsung Galaxy Tab

Of those, only the Galaxy Tab had the then-Android Market (now the Play Store). Hence, if you are only distributing via the Play Store, you might be in position to simply ignore pre-API Level 13 tablets. Use `-swNNNdp` to create your dividing line for larger devices, and the Galaxy Tab will simply use the layouts for your smaller devices.

If this concerns you, or you are also supporting the Kindle Fire and early NOOKs, you can use layout aliases to minimize code duplication. For example, suppose that you have a `res/layout/main.xml` that you wanted to have different versions for phones and tablets, and you want to use `-swNNNdp` for your dividing line as to where the tablet layouts get used, but you also want to have the older tablets, like the Galaxy Tab, use the following recipe:

- Put your tablet-sized layouts in `res/layout/`, but with *different* filenames (e.g., `res/layout/main_to_be_used_for_tablets.xml`)
- In `res/values-swNNNdp/layouts.xml`, for your chosen value of `NNN`, put aliases (via `<item>` elements) for the original names (via the `name` attribute) pointing to the resources you want to use for `-swNNNdp` devices:

```
<resources>
    <item name="main" type="layout">@layout/main_to_be_used_for_tablets</item>
</resources>
```

- In `res/values-large/layouts.xml`, put those same aliases

Now, both older and newer devices, when referencing the same resource name, will get routed to the right layouts for their screen size.

Coping with Complexity

Where things start to get complicated is when you need to use multiple disparate criteria for your resources.

For example, suppose that you have drawable resources that are locale-dependent, such as a stop sign. You might want to have resource sets of drawables tied to language, so you can substitute in different images for different locales. However, you might also want to have those images vary by density, using higher-resolution images on higher-density devices, so the images all come out around the same physical size.

To do that, you would wind up with directories with multiple resource set qualifiers, such as:

- `res/drawable-ldpi/`
- `res/drawable-mdpi/`
- `res/drawable-hdpi/`
- `res/drawable-xhdpi/`

- `res/drawable-en-rUK-ldpi/`
- `res/drawable-en-rUK-mdpi/`
- `res/drawable-en-rUK-hdpi/`
- `res/drawable-en-rUK-xhdpi/`
- And so on

(with the default language being, say, US English, using a US stop sign)

Once you get into these sorts of situations, though, a few rules come into play, such as:

1. The configuration options (e.g., `-en`) have a particular order of precedence, and they must appear in the directory name in that order. The Android documentation outlines the specific order in which these options can appear. For the purposes of this example, screen size is more important than screen orientation, which is more important than screen density, which is more important than whether or not the device has a keyboard.
2. There can only be one value of each configuration option category per directory.
3. Options are case sensitive

For example, you might want to have different layouts based upon screen size and orientation. Since screen size is more important than orientation in the resource system, the screen size would appear in the directory name ahead of the orientation, such as:

- `res/layout-sw600dp-land/`
- `res/layout-sw600dp/`
- `res/layout-land/`
- `res/layout/`

Choosing The Right Resource

Given that you can have N different definitions of a resource, how does Android choose the one to use?

First, Android tosses out ones that are specifically invalid. So, for example, if the language of the device is `-ru`, Android will ignore resource sets that specify other languages (e.g., `-zh`). The exceptions to this are density qualifiers and screen size qualifiers — we will get to those exceptions later.

Then, Android chooses the resource set that has the desired resource and has *the most important distinct qualifier*. Here, by "most important", we mean the one that appears left-most in the directory name, based upon the directory naming rules discussed above. And, by "distinct", we mean where no other resource set has that qualifier.

If there is no specific resource set that matches, Android chooses the default set — the one with no suffixes on the directory name (e.g., `res/layout/`).

With those rules in mind, let's look at some scenarios, to cover the base case plus the aforementioned exceptions.

Scenario #1: Something Simple

Let's suppose that we have a `main.xml` file in:

- `res/layout-land/`
- `res/layout/`

When we call `setContentView(R.layout.main)`, Android will choose the `main.xml` in `res/layout-land/` if the device is in landscape mode. That particular resource set is valid in that case, and it has the most important distinct qualifier (`-land`). If the device is in portrait mode, though, the `res/layout-land/` resource set does not qualify, and so it is tossed out. That leaves us with `res/layout/`, so Android uses that `main.xml` version.

Scenario #2: Disparate Resource Set Categories

It is possible, though bizarre, for you to have a project with `main.xml` in:

- `res/layout-en/`
- `res/layout-land/`
- `res/layout/`

In this case, if the device's locale is set to be English, Android will choose `res/layout-en/`, regardless of the orientation of the device. That is because `-en` is a more important resource set qualifier — "Language and region" appears higher in the "Table 2. Configuration qualifier names" from the Android documentation than does "Screen orientation" (for `-land`). If the device is not set for English, though, Android will toss out that resource set, at which point the decision-making process is the same as in Scenario #1 above.

Scenario #3: Multiple Qualifiers

Now let's envision a project with `main.xml` in:

- `res/layout-en/`
- `res/layout-land-v11/`
- `res/layout/`

You might think that `res/layout-land-v11/` would be the choice, as it is more specific, matching on two resource set qualifiers versus the one or none from the other resource sets.

(in fact, the author of this book thought this was the choice for many years)

In this case, though, language is more important than either screen orientation or Android API level, so the decision-making process is similar to Scenario #2 above: Android chooses `res/layout-en/` for English-language devices, `res/layout-land-v11/` for landscape API Level 11+ devices, or `res/layout/` for everything else.

Scenario #4: Multiple Qualifiers, Revisited

Let's change the resource mix, so now we have a project with `main.xml` in:

- `res/layout-land-night/`
- `res/layout-land-v11/`
- `res/layout/`

Here, while `-land` is the most important resource set qualifier, it is not distinct — we have more than one resource set with `-land`. Hence, we need to check which is the next-most-important resource set qualifier. In this case, that is `-night`, as night mode is a more important category than is Android API level, and so Android will choose `res/layout-land-night/` if the device is in night mode. Otherwise, it will choose `res/layout-land-v11/` if the device is running API Level 11 or higher. If the device is not in night mode and is not running API Level 11 or higher, Android will go with `res/layout/`.

Scenario #5: Screen Density

Now, let's look at the first exception to the rules: screen density.

Android will *always* accept a resource set that contains a screen density, *even if it does not match the density of the device*. If there is an exact density match, of course, Android uses it. Otherwise, it will use what it feels is the next-best match, based upon how far off it is from the device's actual density and whether the other density is higher or lower than the device's actual density.

The reason for this is that for drawable resources, Android will downsample or upsample the image automatically, so the drawable will appear to be the right size, even though you did not provide an image in that specific density.

The catch is two-fold:

1. Android applies this logic to all resources, not just drawables, so even if there is no exact density match on, say, a layout, Android will still choose a resource from another density bucket for the layout
2. As a side-effect of the previous bullet, if you include a density resource set qualifier, Android will ignore any lower-priority resource set qualifiers (unless there are multiple directories with the same density resource set qualifier, in which case the lower-priority qualifiers serve as the "tiebreaker")

So, now let's pretend that our project has `main.xml` in:

- `res/layout-mdpi/`
- `res/layout-nonav/`
- `res/layout/`

Android will choose `res/layout-mdpi/`, even for `-hdpi` devices that do not have a "non-touch navigation method". While `-mdpi` does not match `-hdpi`, Android will still choose `-mdpi`. If we were dealing with drawables resources, Android would upsample the `-mdpi` image.

Scenario #6: Screen Sizes

If you have resource sets tied to screen size, Android will choose the one that is closest to the actual screen size yet smaller than the actual screen size. Resource sets for screen sizes larger than the actual screen size are ignored.

This works for `-swNNNdp`, `-wNNNdp`, and `-hNNNdp` for all devices. On `-large` or `-xlarge` devices, Android applies the same logic for the classic screen size qualifiers (`-small`, `-normal`, `-large`, `-xlarge`). However, Android does *not* apply this logic for `-small` or `-normal` devices — a `-normal` device will not load a `-small` resource.

Now let's pretend that our project has `main.xml` in:

- `res/layout-normal/`
- `res/layout-land/`
- `res/layout/`

Android will choose `res/layout-normal/` if the device is not `-small`. Otherwise, Android will choose `res/layout-land/` if the device is landscape. If all else fails, Android will choose `res/layout/`.

Similarly, if we have:

- `res/layout-w320dp/`
- `res/layout-land/`
- `res/layout/`

Android will choose `res/layout-w320dp/` for devices whose current screen width is 320dp or higher. Otherwise, Android will choose `res/layout-land/` if the device is landscape. If all else fails, Android will choose `res/layout/`.

API-Versioned Resources

As noted previously in this chapter, the `-vNNN` set of suffixes indicate that the resources in that directory are for the stated API level or higher. So, for example, `res/values-v21/` indicates that the resources in that directory should only be used on API Level 21 (Android 5.0) and higher. Devices running older versions of Android will ignore those resources.

This is a particularly important set of suffixes for dealing with major Android version changes. The look and feel of a stock Android app changed significantly at API Level 11 (Android 3.0) and API Level 21 (Android 5.0). You may find that you want to have different resources starting at those API level split points, so that your UI looks appropriate on all versions of Android that you are supporting.

Use Case: Themes by API Level

One big use case for this feature is having different themes by API level.

Even if your `minSdkVersion` is 11 or higher, you may want to have two different themes for your app:

- One, used from API Level 11-20, based on `Theme.Holo`
- Another, used from API Level 21 onwards, based on `Theme.Material`

Your rough alternative is to use the `appcompat-v7` backport of the action bar and bits of the Material Design aesthetic. For highly stylized apps, or in cases where you are sure that you want Material Design on pre-Android 5.0 devices, `appcompat-v7` is worth considering. But if you want to blend in better on each major native UI variant, you will want to support `Theme.Holo` on Android 3.x and 4.x and `Theme.Material` after that.

The hard work here is setting up your themes themselves, such as what was outlined back in the chapter on the action bar. Having them both be available, depending upon device version, is merely a matter of putting the resources into the proper directories.

For example, take a look at the `ActionBar/VersionedColor` sample project. This is a "mashup" of the `HoloColor` and `MaterialColor` sample projects, where the determination of which theme to use is based on API level.

In the `res/values/` directory, we have a `styles.xml` file that is the same as the one in the `HoloColor` example, just with the filename standardized to `styles.xml`. It uses a custom theme (`Theme.Apptheme`) generated by the Action Bar Style Generator.

There is also a `res/values-v21/` directory, indicating values resources to be used on API Level 21 and higher. It has the theme originally seen in the `MaterialColor` example, where the style resource is renamed to `Theme.Apptheme`, to match the one defined in `res/values/`.

Then, with `<application>` referencing `Theme.Apptheme`, we get the right action bar on the right device.

Here, having the style resources names be the same is important, as we are referencing the name in the `<application>` element in the manifest. To be able to pull in the right one, we need them both to have the same name. However, resources that are referred to by only one of those themes, such as color and drawable resources, could go in a versioned directory or not, as you see fit. They *have* to go in versioned directories and *have* to have the same names if you want multiple editions where the API level chooses which edition to use.

For example, the `Theme.Material`-based theme defined in `res/values-v21/styles.xml` references three color resources. The file for those resources happens to

also be in `res/values-v21/` (`colors.xml`). However, since we are not looking to replace those colors based on API level, the `colors.xml` file could be placed in `res/values/` and work just as well. And, if we *did* want to have different colors by API level, we would need those colors defined in all relevant resource sets, such as both `res/values/` *and* `res/values-v21/`.

Default Change Behavior

When you call methods in the Android SDK that load a resource (e.g., the aforementioned `setContentView(R.layout.main)`), Android will walk through those resource sets, find the right resource for the given request, and use it.

But what happens if the configuration changes *after* we asked for the resource? For example, what if the user was holding their device in portrait mode, then rotates the screen to landscape? We would want a `-land` version of our layouts, if such versions exist. And, since we already requested the resources, Android has no good way of handing us revised resources on the fly... except by forcing us to re-request those resources.

So, this is what Android does, by default, to our foreground activity, when the configuration changes on the fly.

Destroy and Recreate the Activity

The biggest thing that Android does is destroy and recreate our activity. In other words:

- Android calls `onPause()`, `onStop()`, and `onDestroy()` on our original instance of the activity
- Android creates a brand new instance of the same activity class, using the same `Intent` that was used to create the original instance
- Android calls `onCreate()`, `onStart()`, and `onResume()` of the new activity instance
- The new activity appears on the screen

This may seem... invasive. You might not expect that Android would wipe out a perfectly good activity, just because the user flicked her wrist and rotated the screen of her phone. However, this is the only way Android has that guarantees that we will re-request all our resources.

Rebuild the Fragments

If your activity is using fragments, the new instance of the activity will contain the same fragments that the old instance of the activity does. This includes both static and dynamic fragments.

By default, Android destroys and recreates the fragments, just as it destroys and recreates the activities. However, as we will see, we do have an option to tell Android to retain certain dynamic fragment instances — for those, it will have the new instance use the same fragment instances as were used by the old activity, instead of creating new instances from scratch.

Recreate the Views

Regardless of whether or not Android recreates all of the fragments, it will call `onCreateView()` on all of the fragments (plus call `onDestroyView()` on the original set of fragments). In other words, Android recreates all of the widgets and containers, to pour them into the new activity instance.

Retain Some Widget State

Android will hold onto the "instance state" of some of the widgets we have in our activity and fragments. Mostly, it holds onto obviously user mutable state, such as:

- What has been typed into an `EditText`
- Whether a `CompoundButton`, like a `CheckBox` or `RadioButton`, is checked or not
- Etc.

Android will collect this information from the widgets of the old activity instance, carry that data forward to the new activity instance, and update the new set of widgets to have that same state.

However:

- Widgets need to have an ID to have their state saved. If you are inflating the widgets from a layout resource, and the widgets have `android:id` values, you meet this requirement. If, however, you are creating the widgets directly in Java code, those widgets do *not* have an ID. You would need to call `setId()`

to give them an ID or manage the state yourself (using the `onSaveInstanceState()` technique described later in this chapter).

- The ID values need to be unique. If there are several widgets with the same ID, you will run into problems. Usually, the only cases where we have several widgets with the same ID is when those widgets come from an adapter, such as an `ArrayAdapter`. And, usually, those widgets are read-only and do not have any state to save. However, if you attempt putting user-modifiable widgets in the layouts inflated by the adapter, or you otherwise have multiple user-modifiable widgets with the same ID, you will need to manage the state yourself (again, using the `onSaveInstanceState()` technique described later in this chapter).

State Saving Scenarios

When the user rotates the screen, or puts the device in a car dock, or changes the language of the device, your process is not terminated. Your foreground activity will be re-created by default — as will your widgets and fragments — but the process sticks around.

However, there are plenty of cases when your process will be terminated once you move into the background. That might be done automatically by Android or manually by the user.

Depending on *how* your process is terminated, there may be ways that the user can return to your app and expect that they will return to it just how they left it. For example, suppose the user is in your app, then presses HOME to move your app to the background. Hours pass, and Android terminates your process to free up memory for other apps. Sometime after that, the user brings up the recent-tasks list and taps on your app in that list. From the *user's* perspective, they should be returning to your app in the same state that they left it when they pressed HOME. However, if your process was terminated, by default you lost all that state.

Some of the techniques for dealing with a configuration change — those involving the "saved instance state `Bundle`" — will *also* help you handle the recent-tasks-list scenario. Some of the other techniques — such as retaining a fragment — only help with handling configuration changes and will do nothing for you in terms of the recent-tasks-list scenario. The general rule of thumb, therefore, is to use the `Bundle` where you can, and use other techniques (e.g., retained fragments) where the `Bundle` is inappropriate or inadequate. We will see those techniques in the next section.

However, bear in mind that all of this state is designed for transient data, data that the user will not mind if they never see again. For example, suppose the user is in your app, then presses HOME to move your app to the background. Hours pass, and due to the user having busily used their device, *you "fall off" the recent-tasks list*, as that list will not extend indefinitely. In this case, if the user starts up your app again (e.g., via the home screen launcher icon), you will *not* get any state information back for use. Data that the user filled into the old app instance, where that data *must be remembered* and reused in any future run of your app, will need to be persisted yourself, in a database or other type of file.

With all of that in mind, let's examine our options for dealing with the transient state, with an emphasis on configuration changes.

Your Options for Configuration Changes

As noted, a configuration change is fairly invasive on your activity, replacing it outright with all new content (albeit with perhaps some information from the old activity's widgets carried forward into the new activity's widgets).

Hence, you have several possible approaches for handling configuration changes in any given activity.

Do Nothing

The easiest thing to do, of course, is to do nothing at all. If all your state is bound up in stuff Android handles automatically, you do not need to do anything more than the defaults.

For example, the `ViewPager/Fragments` demo from the preceding chapter works correctly "out of the box". All of our "state" is tied up in `EditText` widgets, which Android handles automatically. So, we can type in stuff in a bunch of those widgets, rotate the screen (e.g., via ^[Ctrl>-<Right]^ in the emulator on a Windows or Linux PC), and our entered text is retained.

Alas, there are plenty of cases where the built-in behavior is either incomplete or simply incorrect, and we will need to do more work to make sure that our configuration changes are handled properly.

Retain Your Fragments

One approach for handling these sorts of configuration changes is to have Android retain a dynamic fragment.

Here, "retain" means that Android will keep the same fragment instance across the configuration change, detaching it from the original hosting activity and attaching it to a new hosting activity. Since it is the same fragment instance, anything contained inside that instance is itself retained and, therefore, is not lost when the activity is destroyed and recreated.

To see this in action, take a look at the `ConfigChange/Fragments` sample project.

The business logic for this demo (and for all the other demos in this chapter) is that we want to allow the user to pick a contact out of the roster of contacts found on their device or emulator. We will do that by having the user press a "Pick" button, at which time we will display an activity that will let the user pick the contact and return the result to us. Then, we will enable a "View" button, and let the user view the details of the selected contact. The key is that our selected contact needs to be retained across configuration changes — otherwise, the user will rotate the screen, and the activity will appear to forget about the chosen contact.

The activity itself just loads the dynamic fragment, following the recipe seen previously in this book:

```
package com.commonsware.android.rotation.frag;

import android.app.Activity;
import android.os.Bundle;

public class RotationFragmentDemo extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager().beginTransaction()
                                  .add(android.R.id.content,
                                       new RotationFragment()).commit();
    }
  }
}
```

(from ConfigChange/Fragments/app/src/main/java/com/commonsware/android/rotation/frag/RotationFragmentDemo.java)

The reason for checking for the fragment's existence should now be clearer. Since Android will automatically recreate (or retain) our fragments across configuration changes, we do not want to create a *second* copy of the same fragment when we already have an existing copy.

The fragment is going to use an `R.layout.main` layout resource, with two implementations. One, in `res/layout-land/`, will be used in landscape:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/view"
    android:enabled="false"
  />
</LinearLayout>
```

(from ConfigChange/Fragments/app/src/main/res/layout-land/main.xml)

The portrait edition, in `res/layout/`, is identical save for the orientation of the `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
```

```
    android:text="@string/pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/view"
    android:enabled="false"
  />
</LinearLayout>
```

(from ConfigChange/Fragments/app/src/main/res/layout/main.xml)

Here is the complete implementation of `RotationFragment`:

```
package com.commonsware.android.rotation.frag;

import android.app.Activity;
import android.app.Fragment;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class RotationFragment extends Fragment implements
    View.OnClickListener {
  static final int PICK_REQUEST=1337;
  Uri contact=null;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    setRetainInstance(true);

    View result=inflater.inflate(R.layout.main, parent, false);

    result.findViewById(R.id.pick).setOnClickListener(this);

    View v=result.findViewById(R.id.view);

    v.setOnClickListener(this);
    v.setEnabled(contact != null);
```

```
    return(result);
  }

  @Override
  public void onActivityResult(int requestCode, int resultCode,
                               Intent data) {
    if (requestCode == PICK_REQUEST) {
      if (resultCode == Activity.RESULT_OK) {
        contact=data.getData();
        getView().findViewById(R.id.view).setEnabled(true);
      }
    }
  }

  @Override
  public void onClick(View v) {
    if (v.getId() == R.id.pick) {
      pickContact(v);
    }
    else {
      viewContact(v);
    }
  }

  public void pickContact(View v) {
    Intent i=
        new Intent(Intent.ACTION_PICK,
                   ContactsContract.Contacts.CONTENT_URI);

    startActivityForResult(i, PICK_REQUEST);
  }

  public void viewContact(View v) {
    startActivity(new Intent(Intent.ACTION_VIEW, contact));
  }
}
```

(from ConfigChange/Fragments/app/src/main/java/com/commonsware/android/rotation/frag/RotationFragment.java)

In `onClick()`, we hook up the "Pick" button to a `pickContact()` method. There, we call `startActivityForResult()` with an `ACTION_PICK` `Intent`, indicating that we want to pick something from the `ContactsContract.Contacts.CONTENT_URI` collection of contacts. We will discuss `ContactsContract` in greater detail later in this book. For the moment, take it on faith that Android has such an `ACTION_PICK` activity, one that will display to the user the list of available contacts:

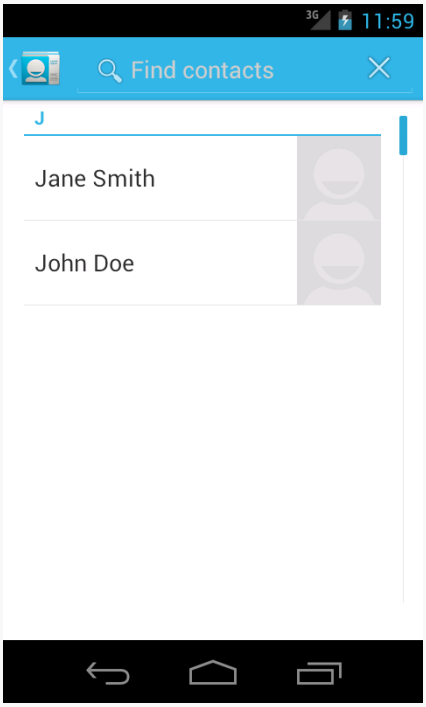


Figure 248: ACTION_PICK of a Contact

In addition to the `ACTION_PICK` `Intent`, we also supply a unique `int` to `startActivityForResult()`. This `int` should be a 16-bit value (`0` to `65535`) that is unique for this particular `startActivityForResult()` call from this activity. This value does not have to be unique compared to such calls from other activities of yours, let alone across the entire device.

If the user picks a contact, control returns to our activity, with a call to `onActivityResult()`. `onActivityResult()` is passed:

- the unique ID we supplied to `startActivityForResult()`, to help identify this result from any others we might be receiving
- `RESULT_OK` if the user did pick a contact, or `RESULT_CANCELED` if the user abandoned the pick activity
- an `Intent` containing the result from the pick activity, which, in this case, will contain a `Uri` representing the selected contact, retrieved via `getData()`

We store that `Uri` in a data member, plus we enable the "View" button, which, when clicked, will bring up an `ACTION_VIEW` activity on the selected contact via its `Uri`:

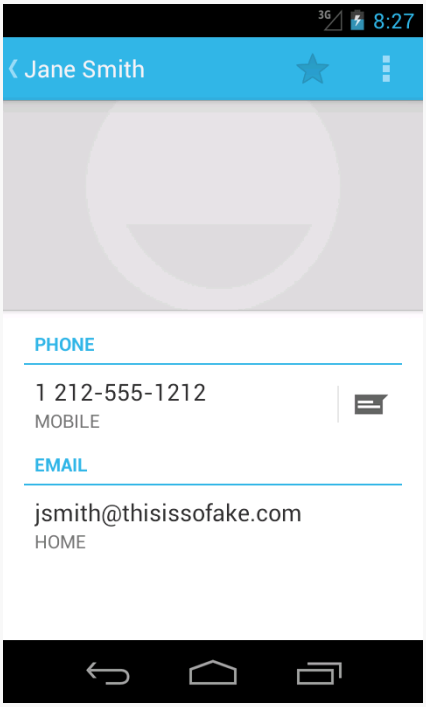


Figure 249: ACTION_VIEW of a Contact

Up in `onCreateView()`, we called `setRetainInstance(true)`. This tells Android to keep this fragment instance across configuration changes. Hence, we can pick a contact in portrait mode, then rotate the screen (e.g., ^[Ctrl>-<Right]^ in the emulator on Windows or Linux), and view the contact in landscape mode. Even though the activity and the buttons were replaced as a result of the rotation, the fragment was not, and the fragment held onto the `Uri` of the selected contact.

Note that `setRetainInstance()` only works with dynamic fragments, not static fragments. Static fragments are always recreated when the activity is itself destroyed and recreated.

The benefit of this technique, over others, is that you can retain any sort of data you want: any data type, any size, etc. However, this approach does not save state that will be given back to you after your process had been terminated, such as when the user goes back to your app via the recent-tasks list.

Model Fragment

A variation on this theme is the "model fragment". While fragments normally are focused on supplying portions of the UI to a user, that is not really a requirement. A

model fragment is one that simply uses `setRetainInstance(true)` to ensure that it sticks around as configurations change. This fragment then holds onto any model data that its host activity needs, so as that activity gets destroyed and recreated, the model data sticks around in the model fragment.

This is particularly useful for data that might not otherwise have a fragment home. For example, imagine an activity whose UI consists entirely of a `ViewPager`, like the tutorial app. Even though that `ViewPager` might hold fragments, there will be many pages in most pagers. It may be simpler to add a separate, UI-less model fragment and have it hold the activity's data model for the `ViewPager`. This allows the activity to still be destroyed and recreated, and even allows the `ViewPager` to be destroyed and recreated, while still retaining the already-loaded data.

Google recommends using a model fragment instead of using `setRetainInstance(true)` with a regular fragment. The less the retained fragment holds, the less likely it is that you will hold something that you should not be holding, such as a string that needs to be reloaded from a string resource due to a possible locale change. That being said, if you are careful and make sure that all your data members are accounted for properly, using `setRetainInstance(true)` from any fragment can be made safe.

Add to the Bundle

If you want state to be maintained not only for configuration changes, but also for process terminations, you will want to use `onSaveInstanceState()` and `onRestoreInstanceState()`.

You can override `onSaveInstanceState()` in your activity. It is passed a `Bundle`, into which you can store data that should be maintained across the configuration change. The catch is that while `Bundle` looks a bit like it is a `HashMap`, it actually cannot hold arbitrary data types, which limits the sort of information you can retain via `onSaveInstanceState()`. `onSaveInstanceState()` is called around the time of `onPause()` and `onStop()`.

The widget state maintained automatically by Android is via the built-in implementation of `onSaveInstanceState()`. If you override it yourself, typically you will want to chain to the superclass to get this inherited behavior, in addition to putting things into the `Bundle` yourself.

That `Bundle` is passed back to you in two places:

- onCreate()
- onRestoreInstanceState()

Since onCreate() is called in many cases other than due to a configuration change, frequently the passed-in Bundle is null. onRestoreInstanceState(), on the other hand, is only called when there is a Bundle to be used.

To see how this works, take a look at the ConfigChange/Bundle sample project.

Here, RotationBundleDemo is an activity with all the same core business logic as was in our fragment in the preceding demo. Since the activity will be destroyed and recreated on a configuration change, we override onSaveInstanceState() and onRestoreInstanceState() to retain our contact, if one was selected prior to the configuration change:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  if (contact != null) {
    outState.putParcelable("contact", contact);
  }
}

@Override
protected void onRestoreInstanceState(Bundle state) {
  super.onRestoreInstanceState(state);

  contact=state.getParcelable("contact");
  viewButton.setEnabled(contact != null);
}
```

(from ConfigChange/Bundle/app/src/main/java/com/commonsware/android/rotation/bundle/RotationBundleDemo.java)

Here, we use putParcelable() to put the Uri into the Bundle. Parcelable is an interface that you can implement on a Java class to allow an instance of it to be put into a Bundle. Uri happens to implement Parcelable. The full details of what Parcelable means and how you can make things implement Parcelable are provided later in this book.

The downside of this approach is that not everything can go into a Bundle. A Bundle cannot hold arbitrary data types, so you cannot put a Socket into a Bundle, for example. Also, this Bundle needs to be fairly small, as it is passed across process

boundaries, so you cannot put large objects (e.g., bitmaps) into the `Bundle`. For those cases, you will have to settle for the retained-fragment approach.

Fragments *and* a Bundle

Fragments also have an `onSaveInstanceState()` method that they can override. It works just like the `Activity` equivalent — you can store data in the supplied `Bundle` that will be supplied back to you later on. The biggest difference is that there is no `onRestoreInstanceState()` method — instead, you are handed the `Bundle` in other lifecycle methods:

- `onCreate()`
- `onCreateView()`
- `onViewCreated()`
- `onActivityCreated()`

We can see this in the ConfigChange/FragmentBundle sample project. This is effectively a mashup of the previous two samples: using fragments, but also using `onSaveInstanceState()` instead of `setRetainInstance(true)`.

Our `RotationFragment` now has an `onSaveInstanceState()` method that looks a lot like the one from the `ConfigChange/Bundle` sample's activity:

```
@Override
public void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  if (contact != null) {
    outState.putParcelable("contact", contact);
  }
}
```

(from ConfigChange/FragmentBundle/app/src/main/java/com/commonsware/android/rotation/fragbundle/RotationFragment.java)

Our `onCreateView()` method examines the passed-in `Bundle`, and if it is not `null` tries to obtain our `contact` from it:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                         Bundle state) {
  View result=inflater.inflate(R.layout.main, parent, false);

  result.findViewById(R.id.pick).setOnClickListener(this);
```

```
    View v=result.findViewById(R.id.view);

    v.setOnClickListener(this);

    if (state != null) {
      contact=(Uri)state.getParcelable("contact");
    }

    v.setEnabled(contact != null);

    return(result);
  }
```

(from ConfigChange/FragmentBundle/app/src/main/java/com/commonsware/android/rotation/fragbundle/RotationFragment.java)

This does not allow our fragment to hold onto arbitrary data, the way `setRetainInstance(true)` does. However, as with `onSaveInstanceState()` at the activity level, there are scenarios that `onSaveInstanceState()` handles that retained fragments will not, such as terminating your process due to low memory, yet the user later uses BACK to return to what should have been your activity (and its fragments).

onRetainNonConfigurationInstance()

If you were to go back to early versions of this book, one of the options discussed in this chapter was `onRetainNonConfigurationInstance()`. That approach was deprecated in API Level 11, replaced with retained fragments.

However, in an unprecedented move, Google un-deprecated `onRetainNonConfigurationInstance()` in API Level 24, making it safe for use again.

Despite the really long method name, its use is simple:

- Implement `onRetainNonConfigurationInstance()` on your `Activity`, returning (nearly) any object you want.
- In `onCreate()` of your `Activity`, call `getLastNonConfigurationInstance()`. If it is not `null`, then it is the same object that you returned from `onRetainNonConfigurationInstance()` milliseconds ago, as part of a configuration change. You can cast the `getLastNonConfigurationInstance()` return value to the appropriate type, save it in a field (or whatever), and use it however you were using it in the previous activity instance.

If you need more than one object to be retained this way, simply have `onRetainNonConfigurationInstance()` wrap up the objects in some data structure of your own design.

The ConfigChange/Retain sample project is nearly identical to the `ConfigChange/Bundle` example shown earlier. In this case, the saved instance state `Bundle` was replaced with a retained object. Since `onRetainNonConfigurationInstance()` can return an object of any type, we can just return the `Uri` representing our contact directly:

```
@Override
public Object onRetainNonConfigurationInstance() {
  return(contact);
}
```

(from ConfigChange/Retain/app/src/main/java/com/commonsware/android/rotation/retain/RotationRetainDemo.java)

In `onCreate()`, we set `contact` to be the result of `getLastNonConfigurationInstance()` (casting it to a `Uri`) and update the `viewButton` to reflect whether or not `contact` is `null`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  viewButton=(Button)findViewById(R.id.view);
  contact=(Uri)getLastNonConfigurationInstance();
  viewButton.setEnabled(contact != null);
}
```

(from ConfigChange/Retain/app/src/main/java/com/commonsware/android/rotation/retain/RotationRetainDemo.java)

When the user rotates the screen, the `Uri` is passed from the old activity instance to the new one, and so we remember our previously-selected contact.

Under the covers, retained fragments uses `onRetainNonConfigurationInstance()` and `getLastNonConfigurationInstance()`.

As a result, the limitations on `onRetainNonConfigurationInstance()` are reminiscent to the limitations on retained fragments:

- Do not return any `Views` from `onRetainNonConfigurationInstance()`, as they will be tied to the old activity instance, not the new one

- Do not return any objects that might need to be replaced due to the configuration change (e.g., strings generated from string resources)
- Do not use this to *replace* the saved instance state `Bundle` — which also handles other scenarios, such as process termination — but instead use it for things that cannot go into that `Bundle` due to size or data type

NOTE: If you try using `onRetainNonConfigurationInstance()`, and it shows up with the deprecation strikethrough formatting in your IDE, your `compileSdkVersion` is somewhere between 11 and 23. Raise it to 24 to clear up the deprecation warning.

DIY

In a few cases, even a retained fragment is insufficient, because transferring and re-applying the state would be too complex or too slow. Or, in some cases, the hardware will get in the way, such as when trying to use the `Camera` for taking pictures — a concept we will cover later in this book.

If you are completely desperate, you can tell Android to *not* destroy and recreate the activity on a configuration change... though this has its own set of consequences. To do this:

- Put an `android:configChanges` entry in your `AndroidManifest.xml` file, listing the configuration changes you want to handle yourself versus allowing Android to handle for you
- Implement `onConfigurationChanged()` in your `Activity`, which will be called when one of the configuration changes you listed in `android:configChanges` occurs

Now, for any configuration change you want, you can bypass the whole activity-destruction process and simply get a callback letting you know of the change.

For example, take a look at the `ConfigChange/DIY` sample project.

In `AndroidManifest.xml`, we add the `android:configChanges` attribute to the `<activity>` element, indicating that we want to handle several configuration changes ourselves:

```
    <activity
      android:name="RotationDIYDemo"
      android:configChanges="keyboardHidden|orientation|screenSize"
      android:label="@string/app_name">
      <intent-filter>
```

```
      <action android:name="android.intent.action.MAIN"/>

      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
```

(from ConfigChange/DIY/app/src/main/AndroidManifest.xml)

Many recipes for this will have you handle `orientation` and `keyboardHidden`. However, nowadays, you need to also handle `screenSize` (and, in theory, `smallestScreenSize`), if you have your `android:targetSdkVersion` set to 13 or higher. Note that this will require your build target (e.g., `compileSdkVersion` in Android Studio) to be set to 13 or higher.

Hence, for those particular configuration changes, Android will not destroy and recreate the activity, but instead will call `onConfigurationChanged()`. In the `RotationDIYDemo` implementation, this simply toggles the orientation of the `LinearLayout` to match the orientation of the device:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);

  LinearLayout container=(LinearLayout)findViewById(R.id.container);

  if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
    container.setOrientation(LinearLayout.HORIZONTAL);
  }
  else {
    container.setOrientation(LinearLayout.VERTICAL);
  }
}
```

(from ConfigChange/DIY/app/src/main/java/com/commonsware/android/rotation/diy/RotationDIYDemo.java)

Since the activity is not destroyed during a configuration change, we do not need to worry at all about the `Uri` of the selected contact — it is not going anywhere.

The problem with this implementation is twofold:

1. We are not handling all possible configuration changes. If the user, say, puts the device into a car dock, Android will destroy and recreate our activity, and we will lose our selected contact.
2. We might forget some resource that needs to be changed due to a configuration change. For example, if we start translating the strings used by

the layouts, and we include `locale` in `android:configChanges`, we not only need to update the `LinearLayout` but also the captions of the `Button` widgets, since Android will not do that for us automatically.

It is these two problems that are why Google does not recommend the use of this technique unless absolutely necessary.

Also, bear in mind that this approach does not help at all for retaining state when your process is terminated and the user returns to your app via the recent-tasks list.

Blocking Rotations

No doubt that you have seen some Android applications that simply ignore any attempt to rotate the screen. Many games work this way, operating purely in landscape mode, regardless of how the device is positioned.

To do this, add `android:screenOrientation="sensorLandscape"`, or possibly `android:screenOrientation="sensorPortrait"`, to your manifest. The "sensor" portions of those names indicate that your app can work in regular or "reverse" versions of the orientation (e.g., "regular" landscape is the device rotated 90 degrees counter-clockwise from portrait, while "reverse" landscape is the device rotated 90 degrees clockwise from portrait). On API Level 18+, you could use `userLandscape` or `userPortrait` instead, as those will honor the user's system-level choice of whether to lock screen rotation or not, defaulting to the behavior of `sensorLandscape` or `sensorPortrait` if the user has not locked screen rotation.

Also, if you use this to lock one or more of your activities to a particular orientation, also use the corresponding `<uses-feature>` element in your manifest:

- `<uses-feature android:name="android.hardware.screen.portrait"/>` if you have an activity locked to portrait
- `<uses-feature android:name="android.hardware.screen.landscape"/>` if you have an activity locked to landscape

This element would go as a child of the root `<manifest>` element but outside of the `<application>` element.

The role of this `<uses-feature>` element is to advertise to the Play Store and similar app distribution channels that your app requires certain hardware features. Such channels might block distribution of your app to platforms where your required

orientation is not supported (e.g., portrait activities on a TV-centric device like Android TV).

And Now, a Word From the Android Project View

Earlier in the book, when introducing Android Studio, we saw the Android project view.

One of the reasons why the Android project view was created was to help you manage resources, particularly across various resource sets.

For example, here is a screenshot of the same Android project, but this time with the `values` resources expanded in the tree:

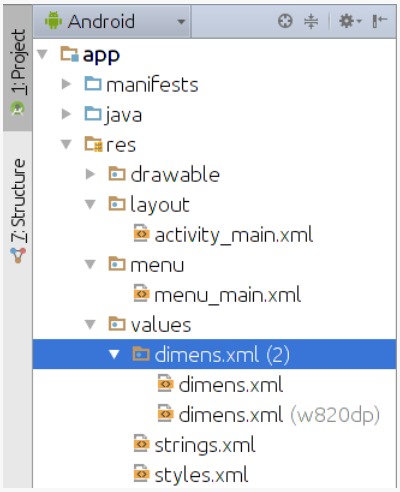


Figure 250: Android Project View, Showing Dimension Resources

The tree makes it appear as though there is just a `res/values/dimens.xml` file... but that the file somehow has children. One child has just the bare `dimens.xml` filename, while the other one has a "(w820dp)" appended.

This reflects the fact that there are two versions of `dimens.xml`: one in `res/values/` and one in `res/values-w820dp/`:

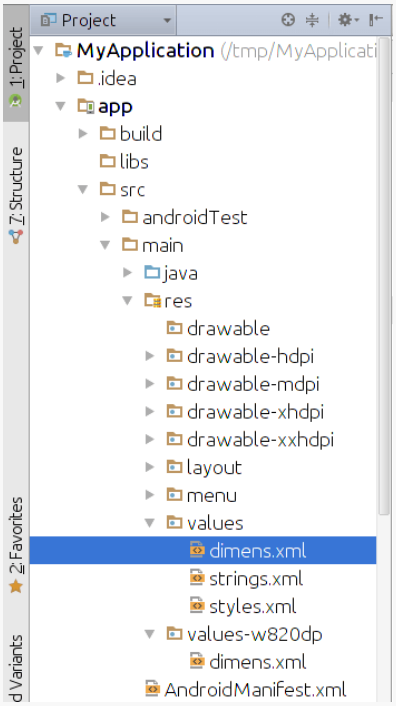


Figure 251: Classic Project View, Showing Dimension Resources

In the Android project view, resources are organized by resource, not by resource set. This can be useful for finding all files that need to be adjusted when you go to adjust one version of the resource, for example.

Configuration Challenges

Some newer Android versions have added new and exciting challenges when dealing with resources and configuration changes.

Multi-Locale Support

Android 7.0+ users can indicate that they support more than one language:

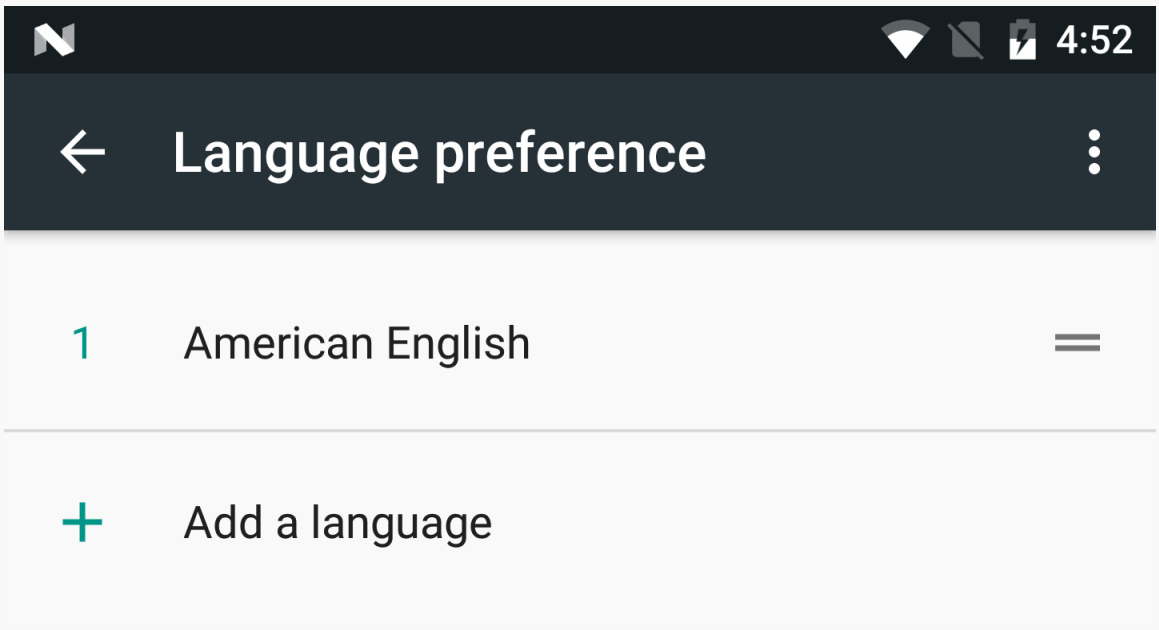


Figure 252: Android 7.0 Language Settings

The user can choose the relative priorities of these languages, by grabbing the handle on the right side of the row and dragging the language higher or lower in the list.

This has impacts on resource resolution for any locale-dependent resources, such as strings. Now Android will check multiple languages for resource matches, before falling back to the default language (e.g., whatever you have in `res/values/strings.xml`). Hence, it is important that you ensure that you have a complete set of strings for every language that you support, lest the user perhaps wind up with a mixed set of languages in the UI.

You can find out what languages the user has requested via a `LocaleList` class and its `getDefault()` `static` method. This, as the name suggests, has a list of `Locale` objects representing the user's preferred languages. If you had previously been using `Locale` alone for this (e.g., for specialized in-app language assistance beyond resources), you will want to switch to `LocaleList` for Android 7.0 and beyond.

Screen Zoom/Dynamic Density

Developers have had the ability to change the effective density of a device, by using `adb` commands. Android 7.0+ users can do this directly from Settings.

In Settings > Display > Display Size, the user can choose five different “display sizes”:

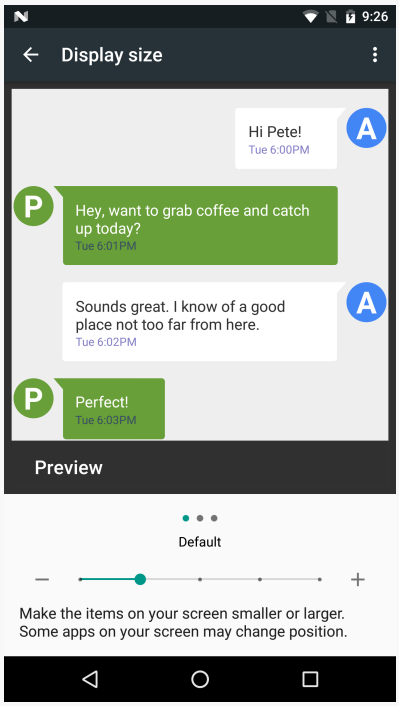


Figure 253: Display Size, Showing Default Setting

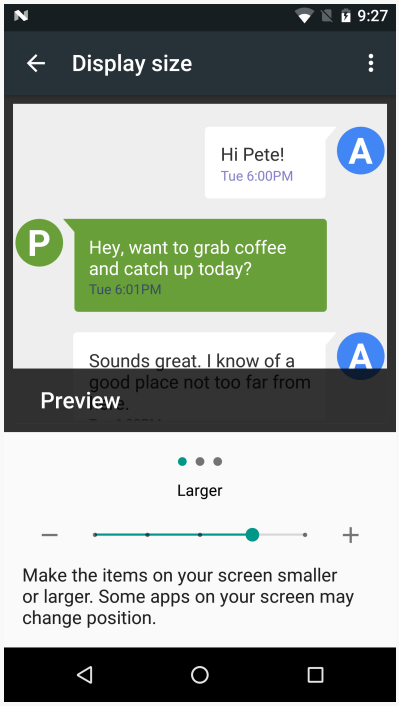


Figure 254: Display Size, Showing "Larger" Setting

While this is described to the user as a "size" or "zoom" setting, in reality it is affecting the apparent screen density, with Android scaling things based on a combination of actual density and the user's stated "display size" preference.

Anything that is already density-independent should work just fine, though you certainly will want to run some tests to ensure that your app is working properly. In fact, this feature makes for a way to easily test your app on a variety of densities, without having to have dedicated hardware for those densities.

If your app is running at the time of the density change, what happens varies:

- Normally, if your `targetSdkVersion` is 23 or below, Android will terminate your process. Since you will be in the background at the time (Settings is in the foreground), this will be little different than if Android terminated your process due to low memory conditions, and your app should already handle this. The documentation suggests that if you have a foreground service, that Android will not terminate the process but instead treat this as a configuration change.
- If your `targetSdkVersion` is 24 or higher, this will be treated as a configuration change for all processes.

For the configuration-change scenarios, if your app is caching information that depends on screen density, be sure to flush those caches and get fresh information based on the new screen density.

Material Design Basics

We have already been exposed to `Theme.Material` as part of this book, such as with the action bar.

Android 5.0+, combined with `Theme.Material`, gives you a lot of capabilities tied to Google's Material Design aesthetic. In this chapter, we will cover some basic Material Design capabilities that will affect your `Theme.Material` app on Android 5.0+, starting with color.

Your App, in Technicolor!

Some developers want to change the colors used by their app to match some specific color or color palette. In some cases, the colors in question are tied to the app's branding. In other cases, the developer simply wants something different than the stock colors you get from something like `Theme.Holo` or `Theme.Holo.Light`.

Affecting color changes in your `Theme.Material`-based Android app is vastly simplified — both for the action bar and the widgets — courtesy of `Theme.Material`'s tinting options.

Basic Tinting Options

In the chapter on the action bar, we saw how to set up a custom theme based on `Theme.Material` that had custom color tinting rules that affected the action bar:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
```

```
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

(from ActionBar/MaterialColor/app/src/main/res/values/styles.xml)

At that time, we focused on the effects that these tints had on the action bar itself. However, with `Theme.Material`, not only do the tints affect the action bar, but they affect the widgets themselves.

The `BasicMaterial` directory contains clones of some of the basic widget samples outlined earlier in this book, where each includes the custom theme demonstrated for the action bar.

In some cases, the custom tints are not normally visible, such as with a button:

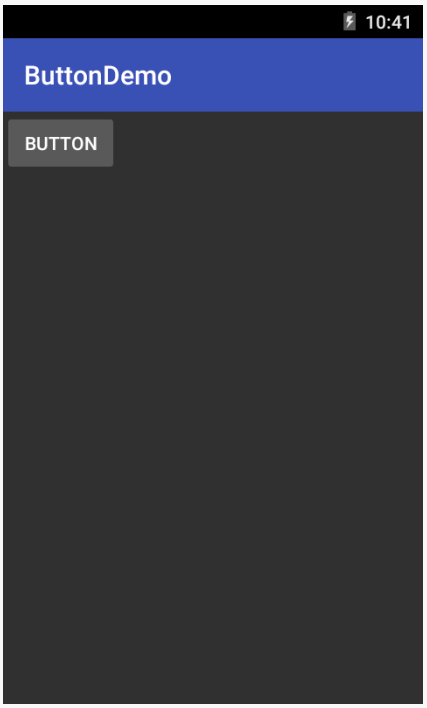


Figure 255: Custom Material Theme for a Button

However, when you tap the button, the animated "ripple" effect shown on the button will use your accent color.

In other cases, the accent color will show up in a more "steady state", such as in a checked `CheckBox`:

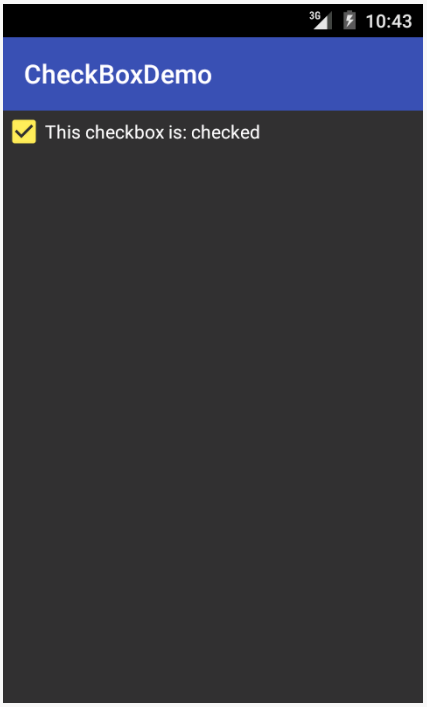


Figure 256: Custom Material Theme for a CheckBox

Similarly, your accent color shows up in things like:

- the "underbar" and drag-cursor in an `EditText`:

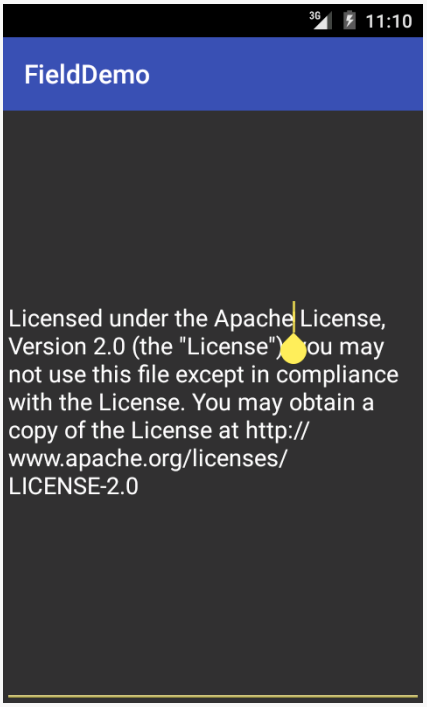


Figure 257: Custom Material Theme for an EditText

- the checked state (and ripple effect when toggling the state) of a `RadioButton`:

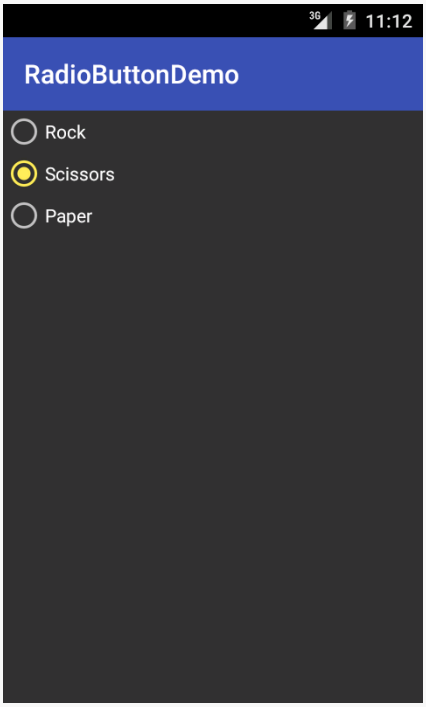


Figure 258: Custom Material Theme for a RadioButton

- and the "checked" state of a `Switch`:

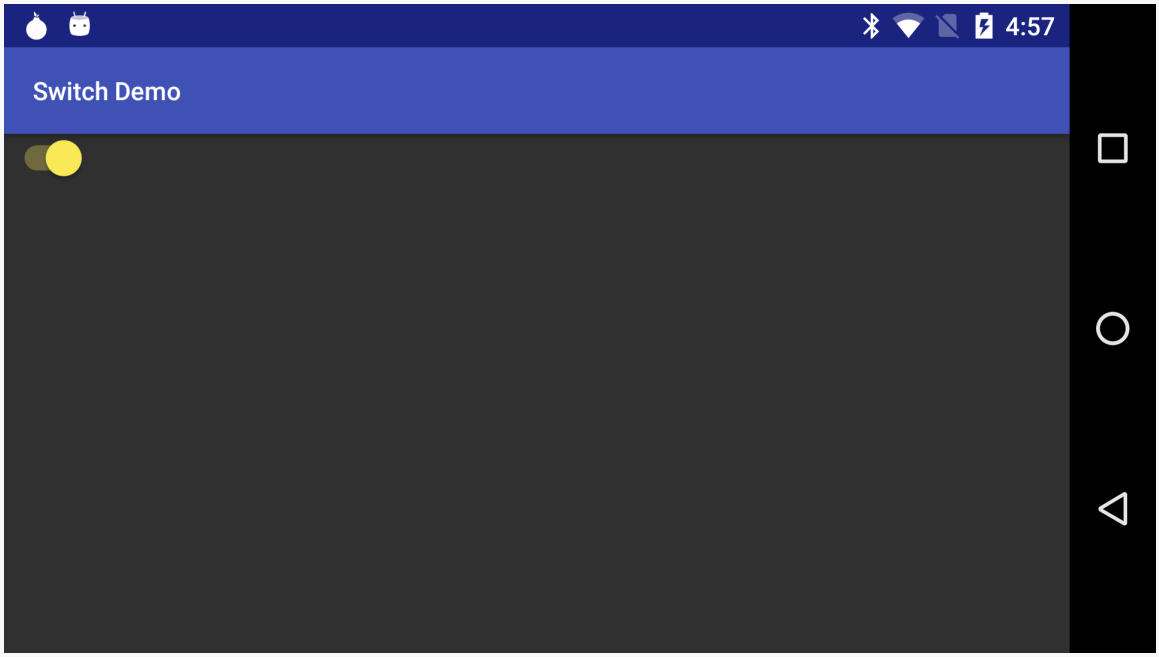


Figure 259: Custom Material Theme for a Switch

Official Google-Approved Colors

Of course, you are welcome to pick whatever colors you like for your theme.

Google has its opinion of what it thinks are good ideas.

As part of the Material Design documentation, you will find a "Color palette" page that outlines possible colors to use.

A Redditor has also published an Android color resource file that contains all of the colors outlined in the Material Design guide.

There is also the material palette site, which generates a color resource file based upon colors that you select from a large grid of color swatches.

Dealing with Threads

Users like snappy applications. Users do not like applications that feel sluggish.

The way to help your application feel snappy is to use the standard threading capabilities built into Android. This chapter will go through the issues involved with thread management in Android and will walk you through some of the options for keeping the user interface crisp and responsive.

The Main Application Thread

When you call `setText()` on a `TextView`, you probably think that the screen is updated with the text you supply, right then and there.

You would be mistaken.

Rather, everything that modifies the widget-based UI goes through a message queue. Calls to `setText()` do not update the screen — they just place a message on a queue telling the operating system to update the screen. The operating system pops these messages off of this queue and does what the messages require.

The queue is processed by one thread, variously called the "main application thread" and the "UI thread". So long as that thread can keep processing messages, the screen will update, user input will be handled, and so on.

However, the main application thread is also used for nearly all callbacks into your activity. Your `onCreate()`, `onClick()`, `onListItemClick()`, and similar methods are all called on the main application thread. While your code is executing in these methods, Android is not processing messages on the queue, and so the screen does not update, user input is not handled, and so on.

This, of course, is bad. So bad, that if you take more than a few seconds to do work on the main application thread, Android may display the dreaded "Application Not Responding" dialog (ANR for short), and your activity may be killed off.

Nowadays, though, the bigger concern is jank.

"Jank", as used in Android, refers to sluggish UI updates, particularly when something is animating. For example, you may have encountered some apps that when you scroll a `ListView` in the app, the `ListView` does not scroll smoothly. Rather, it scrolls jerkily, interleaving periods of rapid movement with periods where the animation is frozen. Most of the time, this is caused by the app's author doing too much work on the main application thread.

Android 4.1 introduced "Project Butter", which, among other things, established a baseline for "doing too much work on the main application thread". We will "drop frames" if we take more than ~16ms per frame (60 frames per second), and dropped frames are the source of jank. Since we may be called many times during a frame, each of our callbacks needs to be very cheap, ideally below 1ms. We will get much more into the issue of jank later in the book, but it is important to understand now that any significant delay in the execution of our code on the main application thread can have visible effects to the user.

Hence, you want to make sure that all of your work on the main application thread happens quickly. This means that anything slow should be done in a background thread, so as not to tie up the main application thread. This includes things like:

1. Internet access, such as sending data to a Web service or downloading an image
2. Significant file operations, since flash storage can be remarkably slow at times
3. Any sort of complex calculations

Fortunately, Android supports threads using the standard `Thread` class from Java, plus all of the wrappers and control structures you would expect, such as the `java.util.concurrent` class package.

However, there is one big limitation: you cannot modify the UI from a background thread. You can only modify the UI from the main application thread. If you call `setText()` on a `TextView` from a background thread, your application will crash, with an exception indicating that you are trying to modify the UI from a "non-UI thread" (i.e., a thread other than the main application thread).

This is a pain.

Getting to the Background

Hence, you need to get long-running work moved into background threads, but those threads need to do something to arrange to update the UI using the main application thread.

There are various facilities in Android for helping with this.

Some are high-level frameworks for addressing this issue for major functional areas. One example of this is the `Loader` framework for retrieving information from databases, and we will examine this in a later chapter.

Sometimes, there are asynchronous options built into other Android operations. For example, when we discuss `SharedPreferences` in a later chapter, we will see that we can persist changes to those preferences synchronously or asynchronously.

And, there are a handful of low-level solutions for solving this problem, ones that you can apply for your own custom business logic.

Asyncing Feeling

One popular approach for handling this threading problem is to use `AsyncTask`. With `AsyncTask`, Android will handle all of the chores of coordinating separate work done on a background thread versus on the UI thread. Moreover, Android itself allocates and removes that background thread. And, it maintains a small work queue, further accentuating the "fire and forget" feel to `AsyncTask`.

The Theory

Theodore Levitt is quoted as saying, with respect to marketing: "People don't want to buy a quarter-inch drill, they want a quarter-inch hole". Hardware stores cannot sell holes, so they sell the next-best thing: devices (drills and drill bits) that make creating holes easy.

Similarly, many Android developers who have struggled with background thread management do not want background threads — they want work to be done off the UI thread, to avoid jank. And while Android cannot magically cause work to not

consume UI thread time, Android can offer things that make such background operations easier and more transparent. `AsyncTask` is one such example.

To use `AsyncTask`, you must:

1. Create a subclass of `AsyncTask`
2. Override one or more `AsyncTask` methods to accomplish the background work, plus whatever work associated with the task that needs to be done on the UI thread (e.g., update progress)
3. When needed, create an instance of the `AsyncTask` subclass and call `execute()` to have it begin doing its work

What you do *not* have to do is:

1. Create your own background thread
2. Terminate that background thread at an appropriate time
3. Call all sorts of methods to arrange for bits of processing to be done on the UI thread

AsyncTask, Generics, and Varargs

Creating a subclass of `AsyncTask` is not quite as easy as, say, implementing the `Runnable` interface. `AsyncTask` uses generics, and so you need to specify three data types:

1. The type of information that is needed to process the task (e.g., URLs to download)
2. The type of information that is passed within the task to indicate progress
3. The type of information that is passed when the task is completed to the post-task code

What makes this all the more confusing is that the first two data types are actually used as varargs, meaning that an array of these types is used within your `AsyncTask` subclass.

This should become clearer as we work our way towards an example.

The Stages of AsyncTask

There are four methods you can override in `AsyncTask` to accomplish your ends.

The one you must override, for the task class to be useful, is `doInBackground()`. This will be called by `AsyncTask` on a background thread. It can run as long as it needs to in order to accomplish whatever work needs to be done for this specific task. Note, though, that tasks are meant to be finite – using `AsyncTask` for an infinite loop is not recommended.

The `doInBackground()` method will receive, as parameters, a varargs array of the first of the three data types listed above — the data needed to process the task. So, if your task's mission is to download a collection of URLs, `doInBackground()` will receive those URLs to process.

The `doInBackground()` method must return a value of the third data type listed above — the result of the background work.

You may wish to override `onPreExecute()`. This method is called, from the UI thread, before the background thread executes `doInBackground()`. Here, you might initialize a `ProgressBar` or otherwise indicate that background work is commencing.

Also, you may wish to override `onPostExecute()`. This method is called, from the UI thread, after `doInBackground()` completes. It receives, as a parameter, the value returned by `doInBackground()` (e.g., success or failure flag). Here, you might dismiss the `ProgressBar` and make use of the work done in the background, such as updating the contents of a list.

In addition, you may wish to override `onProgressUpdate()`. If `doInBackground()` calls the task's `publishProgress()` method, the object(s) passed to that method are provided to `onProgressUpdate()`, but in the UI thread. That way, `onProgressUpdate()` can alert the user as to the progress that has been made on the background work. The `onProgressUpdate()` method will receive a varargs of the second data type from the above list — the data published by `doInBackground()` via `publishProgress()`.

A Sample Task

As mentioned earlier, implementing an `AsyncTask` is not quite as easy as implementing a `Runnable`. However, once you get past the generics and varargs, it is not too bad.

To see an `AsyncTask` in action, this section will examine the `Threads/AsyncTask` sample project.

The Fragment and its AsyncTask

We have a `ListFragment`, named `AsyncDemoFragment`:

```
package com.commonsware.android.async;

import android.support.v4.app.ListFragment;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Toast;
import java.util.ArrayList;

public class AsyncDemoFragment extends ListFragment {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private ArrayList<String> model=new ArrayList<String>();
  private ArrayAdapter<String> adapter=null;
  private AddStringTask task=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);

    task=new AddStringTask();
    task.execute();

    adapter=
      new ArrayAdapter<>(getActivity(),
        android.R.layout.simple_list_item_1,
        model);
  }

  @Override
  public void onViewCreated(View v, Bundle savedInstanceState) {
    super.onViewCreated(v, savedInstanceState);

    getListView().setScrollbarFadingEnabled(false);
    setListAdapter(adapter);
  }
```

```
  @Override
  public void onDestroy() {
    if (task != null) {
      task.cancel(false);
    }

    super.onDestroy();
  }

  class AddStringTask extends AsyncTask<Void, String, Void> {
    @Override
    protected Void doInBackground(Void... unused) {
      for (String item : items) {
        if (isCancelled())
          break;

        publishProgress(item);
        SystemClock.sleep(400);
      }

      return(null);
    }

    @Override
    protected void onProgressUpdate(String... item) {
      if (!isCancelled()) {
        adapter.add(item[0]);
      }
    }

    @Override
    protected void onPostExecute(Void unused) {
      Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
           .show();

      task=null;
    }
  }
}
```

(from Threads/AsyncTask/app/src/main/java/com/commonsware/android/async/AsyncDemoFragment.java)

This is another variation on the *lorem ipsum* list of words, used frequently throughout this book. This time, rather than simply hand the list of words to an `ArrayAdapter`, we simulate having to work to create these words in the background using `AddStringTask`, our `AsyncTask` implementation.

In `onCreate()`, we call `setRetainInstance(true)`, so Android will retain this fragment across configuration changes, such as a screen rotation. Since our fragment is being newly created, we initialize our `model` to be an `ArrayList` of `String` values, plus kick off our `AsyncTask` (the `AddStringTask` inner class, described below), saving the `AddStringTask` in a `task` data member. Then, in `onViewCreated()`, we set up the adapter and attach it to the `ListView`, also preventing the `ListView` scrollbars from fading away as is their norm.

In the declaration of `AddStringTask`, we use the generics to set up the specific types of data we are going to leverage. Specifically:

1. We do not need any configuration information in this case, so our first type is `Void`
2. We want to pass each string "generated" by our background task to `onProgressUpdate()`, so we can add it to our list, so our second type is `String`
3. We do not have any results, strictly speaking (beyond the updates), so our third type is `Void`

The `doInBackground()` method is invoked in a background thread. Hence, we can take as long as we like. In a production application, we would be, perhaps, iterating over a list of URLs and downloading each. Here, we iterate over our static list of *lorem ipsum* words, call `publishProgress()` for each, and then sleep 400 milliseconds to simulate real work being done. We also call `isCancelled()` on each pass, to see if our task has been cancelled, skipping the work if it has so we can clean up this background thread.

Since we elected to have no configuration information, we should not need parameters to `doInBackground()`. However, the contract with `AsyncTask` says we need to accept a varargs of the first data type, which is why our method parameter is `Void...`.

Since we elected to have no results, we should not need to return anything. Again, though, the contract with `AsyncTask` says we have to return an object of the third data type. Since that data type is `Void`, our returned object is `null`.

The `onProgressUpdate()` method is called on the UI thread, and we want to do something to let the user know we are progressing on loading up these strings. In this case, we simply add the string to the `ArrayAdapter`, so it gets appended to the end of the list. However, we only do this if we have not already been canceled.

The `onProgressUpdate()` method receives a `String...` varargs because that is the second data type in our class declaration. Since we are only passing one string per call to `publishProgress()`, we only need to examine the first entry in the varargs array.

The `onPostExecute()` method is called on the UI thread, and we want to do something to indicate that the background work is complete. In a real system, there may be some `ProgressBar` to dismiss or some animation to stop. Here, we simply raise a `Toast` and set `task` to `null`. We do not need to worry about calling `isCancelled()`, because `onPostExecute()` will not be invoked if our task has been cancelled.

Since we elected to have no results, we should not need any parameters. The contract with `AsyncTask` says we have to accept a single value of the third data type. Since that data type is `Void`, our method parameter is `Void unused`.

To use `AddStringTask`, we simply create an instance and call `execute()` on it. That starts the chain of events eventually leading to the background thread doing its work.

If `AddStringTask` required configuration parameters, we would have not used `Void` as our first data type, and the constructor would accept zero or more parameters of the defined type. Those values would eventually be passed to `doInBackground()`.

Our fragment also has an `onDestroy()` method that calls `cancel()` on the `AsyncTask` if it is still outstanding (`task` is not `null`). This work of cancelling the task and checking to see if the task is cancelled exists for two reasons:

1. Efficiency, as we should skip any serious work that is not needed if our task itself is not needed
2. To avoid a crash if we attempt to raise a `Toast` on a destroyed activity, such as the user launching the activity, then pressing BACK before we complete the background work and display the `Toast`

The Activity and the Results

`AsyncDemo` is an `Activity` with the standard recipe for kicking off an instance of a dynamic fragment:

```
package com.commonsware.android.async;

import android.support.v4.app.FragmentActivity;
```

```
import android.os.Bundle;

public class AsyncDemo extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
      getSupportFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new AsyncDemoFragment()).commit();
    }
  }
}
```

(from Threads/AsyncTask/app/src/main/java/com/commonsware/android/async/AsyncDemo.java)

If you build, install, and run this project, you will see the list being populated in "real time" over a few seconds, followed by a `Toast` indicating completion.

Threads and Configuration Changes

One problem with the default destroy-and-create cycle that activities go through on a configuration change comes from background threads. If the activity has started some background work — through an `AsyncTask`, for example – and then the activity is destroyed and re-created, somehow the `AsyncTask` needs to know about this. Otherwise, the `AsyncTask` might well send updates and final results to the *old* activity, with the new activity none the wiser. In fact, the new activity might start up the background work *again*, wasting resources.

That is why, in the sample above, we are retaining the fragment instance. The fragment instance holds onto its data model (in this case, the `ArrayList` of Latin words) and knows not to kick off a new `AsyncTask` just because the configuration changed. Moreover, we retain that data model, so the new `ListView` created due to the configuration change can work with a new adapter backed by the old data model, so we do not lose our existing set of Latin words.

We also have to be very careful not to try referring to the activity (via `getActivity()` on the fragment) from our background thread (`doInBackground()`). Because, suppose that during the middle of the `doInBackground()` processing, the user rotates the screen. The activity we work with will change on the fly, on the main application thread, independently of the work being done in the background. The activity returned by `getActivity()` may not be in a useful state for us while this configuration change is going on.

However, it is safe for us to use `getActivity()` from `onPostExecute()`, and even from `onProgressUpdate()`. For those callbacks, either the configuration change has not yet happened, or it has been completed — we will not be in the middle of the change.

Where Not to Use AsyncTask

`AsyncTask`, particularly in conjunction with a dynamic fragment, is a wonderful solution for most needs for a background thread.

The key word in that sentence is "most".

`AsyncTask` manages a thread pool, from which it pulls the threads to be used by task instances. Thread pools assume that they will get their threads back after a reasonable period of time. Hence, `AsyncTask` is a poor choice when you do not know how long you need the thread (e.g., thread listening on a socket for a chat client, where you need the thread until the user exits the client).

About the AsyncTask Thread Pool

Moreover, the thread pool that `AsyncTask` manages has varied in size.

In Android 1.5, it was a single thread.

In Android 1.6, it was expanded to support many parallel threads, probably more than you will ever need.

In Android 3.2, it has shrunk back to a single thread, if your `android:targetSdkVersion` is set to 13 or higher. This was to address concerns about:

- Forking too many threads and starving the CPU
- Developers thinking that there is an ordering dependency between forked tasks, when with the parallel execution there is none

If you wish, starting with API Level 11, you can supply your own `Executor` (from the `java.util.concurrent` package) that has whatever thread pool you wish, so you can manage this more yourself. In addition to the serialized, one-at-a-time `Executor`, there is a built-in `Executor` that implements the old thread pool, that you can use rather than rolling your own.

If your `minSdkVersion` is 11 or higher, use `executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)` if you specifically want to opt into a multi-thread thread pool. If your `minSdkVersion` is below 11, you will still want to do that... but only on API Level 11+ devices, falling back to `execute()` on the older devices. This static utility method handles this for you:

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
static public <T> void executeAsyncTask(AsyncTask<T, ?, ?> task,
                                        T... params) {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, params);
  }
  else {
    task.execute(params);
  }
}
```

To use this, call `executeAsyncTask()`, passing in your `AsyncTask` instance and the parameters you would ordinarily have passed to `execute()`.

An explanation of what we are doing here, in terms of the `@TargetApi` annotation and such, will come later in the book.

Also note that the number of threads in the multiple-thread thread pool has *also* changed over the years. Originally, that pool could climb to as many as 128 threads, which was far too many. As of Android 4.4, the thread pool will only grow to "the number of CPU cores * 2 + 1", so on a dual-core device, the thread pool will cap at 5 threads. Further tasks will be queued, up to a maximum of 128 queued tasks.

Alternatives to AsyncTask

There are other ways of handling background threads without using `AsyncTask`:

- You can employ a `Handler`, which has a `handleMessage()` method that will process `Message` objects, dispatched from a background thread, on the main application thread
- You can supply a `Runnable` to be executed on the main application thread to `post()` on any `View`, or to `runOnUiThread()` on `Activity`
- You can supply a `Runnable`, plus a delay period in milliseconds, to `postDelayed()` on any `View`, to run the `Runnable` on the main application thread after *at least* that number of millisecond has elapsed

Of these, the `Runnable` options are the easiest to use.

These can also be used to allow the main application thread to postpone work, to be done *later* on the main application thread. For example, you can use `postDelayed()` to set up a lightweight polling "loop" within an activity, without needing the overhead of an extra thread, such as the one created by `Timer` and `TimerTask`. To see how this works, let's take a peek at the `Threads/PostDelayed` sample project.

This project contains a single activity, named `PostDelayedDemo`:

```
package com.commonsware.android.post;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PostDelayedDemo extends Activity implements Runnable {
  private static final int PERIOD=5000;
  private View root=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    root=findViewById(android.R.id.content);
  }

  @Override
  public void onStart() {
    super.onStart();

    run();
  }

  @Override
  public void onStop() {
    root.removeCallbacks(this);

    super.onStop();
  }

  @Override
  public void run() {
    Toast.makeText(PostDelayedDemo.this, "Who-hoo!", Toast.LENGTH_SHORT)
         .show();
```

```
    root.postDelayed(this, PERIOD);
  }
}
```

(from Threads/PostDelayed/app/src/main/java/com/commonsware/android/post/PostDelayedDemo.java)

We want to display a `Toast` every five seconds. To do this, in `onCreate()`, we get our hands on the container for an activity's UI, known as `android.R.id.content`, via `findViewById()`. Then, in `onStart()`, we call a `run()` method on our activity, which displays the `Toast` and calls `postDelayed()` to schedule *itself* (as an implementation of `Runnable`) to be run again in `PERIOD` milliseconds. While our activity is in the foreground, the `Toast` will appear every `PERIOD` milliseconds as a result. Once something else comes to the foreground — such as by the user pressing BACK — our `onStop()` method is called, where we call `removeCallbacks()` to "undo" the `postDelayed()` call.

And Now, The Caveats

Background threads, while eminently possible using `AsyncTask` and kin, are not all happiness and warm puppies. Background threads not only add complexity, but they have real-world costs in terms of available memory, CPU, and battery life.

To that end, there is a wide range of scenarios you need to account for with your background thread, including:

1. The possibility that users will interact with your activity's UI while the background thread is chugging along. If the work that the background thread is doing is altered or invalidated by the user input, you will need to communicate this to the background thread. Android includes many classes in the `java.util.concurrent` package that will help you communicate safely with your background thread.
2. The possibility that the process will be terminated while your work is still going on. This is why in many cases, rather than use an `AsyncTask` or a bare `Thread`, you will wind up using a `Service`, such as an `IntentService`. This will be explored in greater detail later in this book.
3. The possibility that your user will get irritated if you chew up a lot of CPU time and battery life without giving any payback. Tactically, this means using `ProgressBar` or other means of letting the user know that something is happening. Strategically, this means you still need to be efficient at what you do — background threads are no panacea for sluggish or pointless code.

4. The possibility that you will encounter an error during background processing. For example, if you are gathering information off the Internet, the device might lose connectivity. Alerting the user of the problem via a `Notification` and shutting down the background thread may be your best option.

Event Buses

Event-driven programming has been around for nearly a quarter-century. Much of Android's UI model is event-driven, where we find out about these events via callbacks (e.g., `onCreate()` for the "start an activity" event) and registered listeners (e.g., `OnClickListener` for when the user taps on a widget).

However, originally, Android did not have a very fine-grained event or message bus implementation that we as developers could use. The `Intent` system works like a message bus, but it is aimed at inter-process communication (IPC) as much as in-process communication, and that comes with some costs.

However, over time, particularly starting in 2012, event buses started to pop up, and these are very useful for organizing communication within your Android application and across threads. Used properly, an event bus can eliminate the need for `AsyncTask` and the other solutions for communicating back to the main application thread, while simultaneously helping you logically decouple independent pieces of your code.

What Is an Event Bus?

Whether you consider it an "event bus" (or "message bus"), the "publisher/subscriber" (or "pub/sub") pattern, or a subset of the "observer" pattern, the programming model where components produce events that others consume is reasonably common in modern software development.

An event bus is designed to decouple the sources of events from the consumers of those events. Or, as one event bus author put it:

> I want an easy, centralized way to notify code that's interested in specific types of events when those events occur without any direct coupling between the code the publishes [sic] an event and the code that receives it.

With the traditional Java listener or observer pattern implementation, the component producing an event needs direct access to consumers of that event. Sometimes, that list of consumers is limited to a single consumer, as with many event handlers associated with Android widgets (e.g., just one `OnClickListener`). But this source-holds-the-sinks coding pattern limits flexibility, as it requires explicit registration by consumers with producers of events, and it may not be that easy for the consumer to *reach* the producer. Furthermore, such direct connections are considered to be a relatively strong coupling between those components, and often times our objective is to have looser coupling.

An event bus provides a standard communications channel (or "bus") that event producers and event consumers can hook into. Event producers merely need to hand the event to the bus; the bus will handle directing those events to relevant consumers. This reduces the coupling between the producers and consumers, sometimes even reducing the amount of code needed to source and sink these events.

OK, But Why Are We Bothering With This?

Later on, we are going to have components other than our activities. In particular, we will have services, which are designed to run briefly in the background to perform some operation. Just as communications between activities tends to be loosely coupled, so too are communications between activities and services. An event bus is a great way for the service to let other pieces of the app know that certain work was done (e.g., "the download is complete, so update the UI").

In the short term, we will use an event bus to have a model fragment let the app know that some data was loaded. In the tutorials, "some data" will be the book contents; in the sample app illustrated in this chapter, "some data" will be some Latin words.

Introducing greenrobot's EventBus

The event bus implementation that we will be using in the tutorials is greenrobot's EventBus, an open source implementation based on the Guava project's event bus. With greenrobot's EventBus, it is fairly easy to send a message from one part of your app to another disparate part of your app.

To illustrate its use, take a look at the `EventBus/AsyncDemo3` sample project. This is a reworking of a previous example that used an `AsyncTask` to pretend to download our

list of Latin words, populating a `ListView` with those words as they arrive. This sample replaces the `AsyncTask` with a model fragment that will keep track of the words and a background thread that will "download" the words. We will use events raised by the model fragment to let the UI fragment know words as they arrive.

Requesting the Artifact

greenrobot's EventBus is distributed as an artifact that you can integrate in your project via the `dependencies` in your module's `build.gradle` file:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'org.greenrobot:eventbus:3.0.0'
}

android {
    compileSdkVersion 19
    buildToolsVersion '26.0.2'

  defaultConfig {
    targetSdkVersion 17
    applicationId "com.commonsware.android.eventbus.greenrobot3"
  }
}
```

(from EventBus/AsyncDemo3/app/build.gradle)

Defining Events

With greenrobot's EventBus, the "events" are objects of arbitrary classes that you define. Each different class represents a different type of event, and you can define as many different event classes as you wish. Those classes do not need to inherit from any special base class, or implement some special interface, or have any magic annotations. They are just classes.

You may wish to put data members, constructors, and accessor methods on the event classes, for any data you wish to pass around specific to the event itself. A `SearchEvent`, for example, might include the search query string as part of the event object.

In our case, we have a `WordReadyEvent` that contains the new word:

```
package com.commonsware.android.eventbus;

class WordReadyEvent {
  private String word;

  WordReadyEvent(String word) {
    this.word=word;
  }

  String getWord() {
    return(word);
  }
}
```

(from EventBus/AsyncDemo3/app/src/main/java/com/commonsware/android/eventbus/WordReadyEvent.java)

Posting Events

To post an event, all you need to do is obtain an instance of an `EventBus` – typically via the `getDefault()` method on `EventBus` — and call `post()` on it, passing in the event to be delivered to any interested party within your app.

With that in mind, let's look at the `ModelFragment` that will be loading in our words:

```
package com.commonsware.android.eventbus;

import android.app.Fragment;
import android.os.Bundle;
import android.os.SystemClock;
import org.greenrobot.eventbus.EventBus;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ModelFragment extends Fragment {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private List<String> model=
    Collections.synchronizedList(new ArrayList<String>());
  private boolean isStarted=false;

  @Override
  public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);

    setRetainInstance(true);

    if (!isStarted) {
      isStarted=true;
      new LoadWordsThread().start();
    }
  }

  public ArrayList<String> getModel() {
    return(new ArrayList<String>(model));
  }

  class LoadWordsThread extends Thread {
    @Override
    public void run() {
      for (String item : items) {
        if (!isInterrupted()) {
          model.add(item);
          EventBus.getDefault().post(new WordReadyEvent(item));
          SystemClock.sleep(400);
        }
      }
    }
  }
}
```

(from EventBus/AsyncDemo3/app/src/main/java/com/commonsware/android/eventbus/ModelFragment.java)

This fragment has no UI — it exists solely to manage a data model on behalf of the rest of the hosting activity. Hence, there is no `onCreateView()` or any other UI logic directly in this fragment.

In `onCreate()`, we call `setRetainInstance(true)`, so that if the user rotates the screen or otherwise triggers a configuration change, our model fragment will survive the change and be attached to the new activity instance. Then, if we have not already started the `LoadWordsThread`, we do so. `LoadWordsThread` iterates over our list of words, sleeps for 400ms to simulate doing real work, adds each word to an `ArrayList` of words that it manages... and calls `post()` to raise a `WordReadyEvent` to let something else know that the model has changed.

Receiving Events

To receive posted events, you need to do three things:

1. Call `register()` on the `EventBus` to tell it that you have an object that wants to receive events
2. Call `unregister()` on the `EventBus` to tell it to stop delivering events to a previously-registered object
3. Implement methods annotated with `@Subscribe` to indicate the type of event you want to receive and to actually process those events

This sample app has an `AsyncDemoFragment` that performs those three steps:

```
package com.commonsware.android.eventbus;

import android.app.Activity;
import android.app.ListFragment;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import org.greenrobot.eventbus.EventBus;
import org.greenrobot.eventbus.Subscribe;
import org.greenrobot.eventbus.ThreadMode;
import java.util.ArrayList;

public class AsyncDemoFragment extends ListFragment {
  private ArrayAdapter<String> adapter=null;
  private ArrayList<String> model=null;

  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
    adapter=
        new ArrayAdapter<String>(getActivity(),
                                 android.R.layout.simple_list_item_1,
                                 model);

    getListView().setScrollbarFadingEnabled(false);
    setListAdapter(adapter);
  }

  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);

    EventBus.getDefault().register(this);
  }

  @Override
  public void onDetach() {
    EventBus.getDefault().unregister(this);
```

```
    super.onDetach();
  }

  @Subscribe(threadMode = ThreadMode.MAIN)
  public void onWordReady(WordReadyEvent event) {
    adapter.add(event.getWord());
  }

  public void setModel(ArrayList<String> model) {
    this.model=model;
  }
}
```

(from EventBus/AsyncDemo3/app/src/main/java/com/commonsware/android/eventbus/AsyncDemoFragment.java)

The fragment starts by overriding `onViewCreated()`, where we create an `ArrayAdapter` and use that to populate the `ListView`.

The `onAttach()` and `onDetach()` methods are where we indicate to the `EventBus` that this fragment object wants to receive relevant posted events. `onAttach()` calls `register()`; `onDetach()` calls `unregister()`.

The `onWordReady()` method, via its parameter and `@Subscribe` annotation, indicates that we are interested in `WordReadyEvents` as they are raised. Our `onWordReady()` method will be called for each `WordReadyEvent` passed to `post()` on the `EventBus`. Since the `@Subscribe` annotation has `threadMode = ThreadMode.MAIN` as part of its configuration, `onWordReady()` will be called on the main application thread, so it is safe for us to update our UI. greenrobot's EventBus is responsible for getting this event to the main application thread — note that we are posting the event from the `LoadWordsThread`, which is a background thread.

In `onWordReady()`, we get the newly-added word, which we can add to our `ArrayAdapter`. `add()` on `ArrayAdapter` appends the word to the end of the list and informs the attached `ListView` that the data changed, so the `ListView` can redraw itself.

What is not obvious, though, from the code in this class is how we are getting the `model` that we are using in `onViewCreated()`. `AsyncDemoFragment` has its own `ArrayList` of words, set via the `setModel()` method. Our `ArrayAdapter` is wrapped around this model. But the master copy of the words is being held by the `ModelFragment`. If the `ModelFragment` has the model, and the `AsyncDemoFragment` needs the model, how are the two being connected?

The Activity

That is handled by our hosting activity, as it sets up these two fragments:

```
package com.commonsware.android.eventbus;

import android.app.Activity;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Bundle;

public class AsyncDemo extends Activity {
  private static final String MODEL_TAG="model";
  private ModelFragment mFrag=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    FragmentManager mgr=getFragmentManager();
    FragmentTransaction trans=mgr.beginTransaction();

    mFrag=(ModelFragment)mgr.findFragmentByTag(MODEL_TAG);

    if (mFrag == null) {
      mFrag=new ModelFragment();
      trans.add(mFrag, MODEL_TAG);
    }

    AsyncDemoFragment demo=
        (AsyncDemoFragment)mgr.findFragmentById(android.R.id.content);

    if (demo == null) {
      demo=new AsyncDemoFragment();
      trans.add(android.R.id.content, demo);
    }

    demo.setModel(mFrag.getModel());

    if (!trans.isEmpty()) {
      trans.commit();
    }
  }
}
```

(from EventBus/AsyncDemo3/app/src/main/java/com/commonsware/android/eventbus/AsyncDemo.java)

In `onCreate()`, we first see if we already have an instance of our model fragment, held by the `FragmentManager` under a `MODEL_TAG` tag. If not, we create an instance of the `ModelFragment` and add it to the `FragmentManager`, under that tag, via a `FragmentTransaction`.

We then see if we already have an instance of our `AsyncDemoFragment`. If not, we create one and add it to the `FragmentManager`, pouring its UI into `android.R.id.content`, via another `FragmentTransaction`.

Then, we connect the two, calling `getModel()` on the `ModelFragment` and handing the result to `setModel()` on the `AsyncDemoFragment`.

When our activity is newly launched, neither fragment exists. Both fragments are created, and the `AsyncDemoFragment` gets its model array from the `ModelFragment`. That array is initially empty. As the `ModelFragment` adds elements to the array, it posts the `WordReadyEvent`, which triggers the `AsyncDemoFragment` to tell the `ArrayAdapter` and `ListView` that the model data changed.

If we undergo a configuration change, the `ModelFragment` is retained, but the `AsyncDemoFragment` is not. Hence, the activity will always be creating an `AsyncDemoFragment`. But the model we give to the `AsyncDemoFragment` may already have words in it, and those words will appear immediately when the `ArrayAdapter` is wrapped around the model. If the `LoadWordsThread` is still running, the new `AsyncDemoFragment` will pick up any new `WordReadyEvents` that are raised, triggering it to update the `ListView` as before.

Visit the Trails!

We will cover much more about jank, and how to detect and diagnose it, in a later chapter.

There are many more features in the greenrobot EventBus implementation. We will see some of those, plus other event bus implementations, in a later chapter on event bus alternatives.

Requesting Permissions

In the late 1990's, a wave of viruses spread through the Internet, delivered via email, using contact information culled from Microsoft Outlook. A virus would simply email copies of itself to each of the Outlook contacts that had an email address. This was possible because, at the time, Outlook did not take any steps to protect data from programs using the Outlook API, since that API was designed for ordinary developers, not virus authors.

Nowadays, many applications that hold onto contact data secure that data by requiring that a user explicitly grant rights for other programs to access the contact information. Those rights could be granted on a case-by-case basis or all at once at install time.

Android is no different, in that it requires permissions for applications to read or write contact data. Android's permission system is useful well beyond contact data, and for content providers and services beyond those supplied by the Android framework.

You, as an Android developer, will frequently need to ensure your applications have the appropriate permissions to do what you want to do with other applications' data. This chapter covers this topic, both the classic approach used for all permissions prior to Android 6.0 and the new runtime permission system used for certain permissions in Android 6.0+.

You may also elect to require permissions for other applications to use your data or services, if you make those available to other Android components. This will be discussed later in this book.

Frequently-Asked Questions About Permissions

Permissions are occasionally a confusing topic in Android app development, more so now that Android 6.0 has arrived and has changed the permission system a fair bit. Here are some common questions about permissions to help get us started.

What Is a Permission?

A permission is a way for Android (or, sometimes, a third-party app) to require an app developer to notify the user about something that the app will do that might raise concerns with the user. Only if an app holds a certain permission can the app do certain things that are defended by that permission.

Mechanically, permissions take the form of elements in the manifest. Right now, we are focusing on requesting and holding permissions, and so we will be working with the `<uses-permission>` element.

When Will I Need a Permission?

Most permissions that you will deal with come from Android itself. Usually, the documentation will tell you when you need to request and hold one of these permissions.

However, occasionally the documentation has gaps.

If you are trying out some code and you crash with a `SecurityException` the description of the exception may tell you that you need to hold a certain permission — that means you need to add the corresponding `<uses-permission>` element to your manifest.

Third-party code, including Google's own Play Services SDK, may define their own custom permissions. Once again, ideally, you find out that you need to request a permission through documentation, and otherwise you find out through crashing during testing.

What Are Some Common Permissions, and What Do They Defend?

There are dozens upon dozens of permissions in Android. Here are some of the permissions we will see in this book:

- `INTERNET`, if your application wishes to access the Internet through any means from your own process, using anything from raw Java sockets through the `WebView` widget
- `WRITE_EXTERNAL_STORAGE`, for writing data to external storage
- `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, for determining where the device is
- `READ_CONTACTS`, to get at personally-identifying information of arbitrary contacts that the user has in their Contacts app

In this book and in casual conversation, we refer to the permissions using the unique portion of their name (e.g., `INTERNET`). Really, the full name of the permission will usually have `android.permission.` as a prefix (e.g., `android.permission.INTERNET`), for Android-defined permissions. Custom permissions from third-party apps should use a different prefix. You will need the full permission name, including the prefix, in your manifest entries.

How Do I Request a Permission?

Put a `<uses-permission>` element in your manifest, as a direct child of the root `<manifest>` element (i.e., as a peer element of `<application>`), with an `android:name` attribute identifying the permission that you are interested in.

For example, here is a sample manifest, with a request to hold the `WRITE_EXTERNAL_STORAGE` permission:

```
<?xml version="1.0"?>
<manifest package="com.commonsware.android.fileseditor"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
```

```
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Files/FilesEditor/app/src/main/AndroidManifest.xml)

This is sufficient for most permissions and most devices. Permissions considered to be `dangerous` need special attention on Android 6.0+, and we will cover that in grand detail later in this chapter.

Note that you are welcome to have zero, one, or several such `<uses-permission>` elements. Also note that some libraries that you elect to use might add their own `<uses-permission>` elements to your manifest, through a process called "manifest merger".

When Is the User Informed About These Permissions?

Well, that gets complicated. It depends on the permission, the version of Android the user is using, from where the user is installing the app, and the phase of the moon.

(well, OK, not really that last one)

Installing Through SDK Tools

Anyone who installs an app using Android Studio will not be prompted for permissions. The same holds true for anyone using anything else based on the Android SDK tools — while the app may request permissions, the user is not prompted for them, and the permissions are granted.

(Android 6.0+ and `dangerous` permissions change this up a bit – more on that later in this chapter)

Installing from the Play Store, Android 5.1 and Older

If the user is running an Android 5.1 or older device, and the user goes to install your app from the Play Store, the user will be presented with a roster of permission groups that contain permissions that you are requesting and that are considered to be `dangerous`:

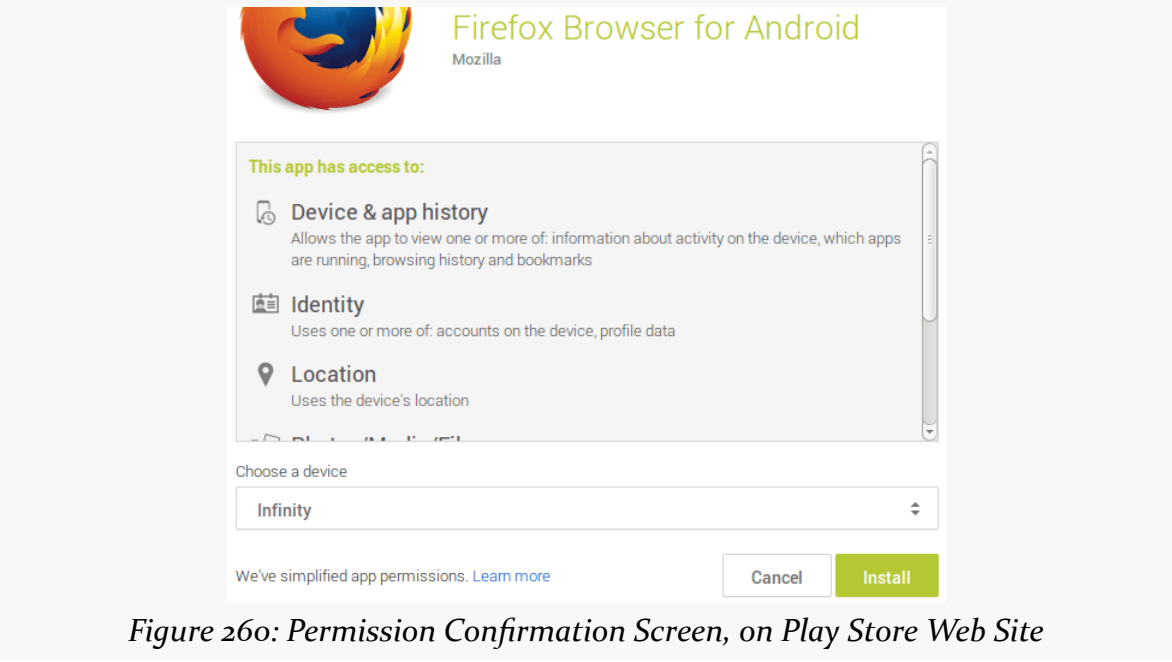


Figure 260: Permission Confirmation Screen, on Play Store Web Site

We will discuss more about permission groups and this `dangerous` concept later in this chapter.

Installing from the Play Store, Android 6.0+

On Android 6.0 and higher, when the user installs your app from the Play Store, what happens depends upon the value of `targetSdkVersion` for your app.

If your `targetSdkVersion` is 22 or lower, you get the same behavior as is described above, where the user sees the list of permission groups which contain permissions that you are requesting and that are considered to be `dangerous`.

If your `targetSdkVersion` is 23 or higher, the user is not prompted about permissions at install time. Instead these prompts will occur when the user runs

your app and when you ask the user for the permissions, as we will see later in this chapter.

Installing by Other Means, Android 5.1 and Older

If you install an app on Android 5.1 or older, by any means (e.g., downloading from a Web site), you will be prompted with a list of all requested permissions:

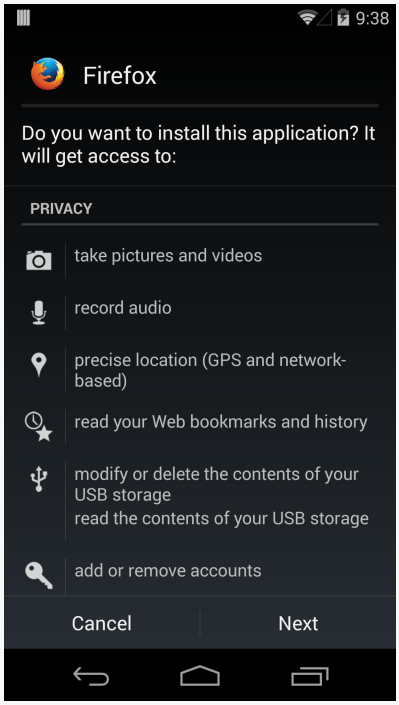


Figure 261: Permission Confirmation Screen, on Android 4.4

Note that this prompt will not appear until you actually have downloaded the app and have begun the installation process. Before then, the device cannot examine the manifest inside the APK file to find the permissions.

Installing by Other Means, Android 6.0+

If your app's `targetSdkVersion` is 23 or higher, and you install the app on an Android 6.0+ device by other means than the Play Store, you will not be prompted about any permissions at install time:

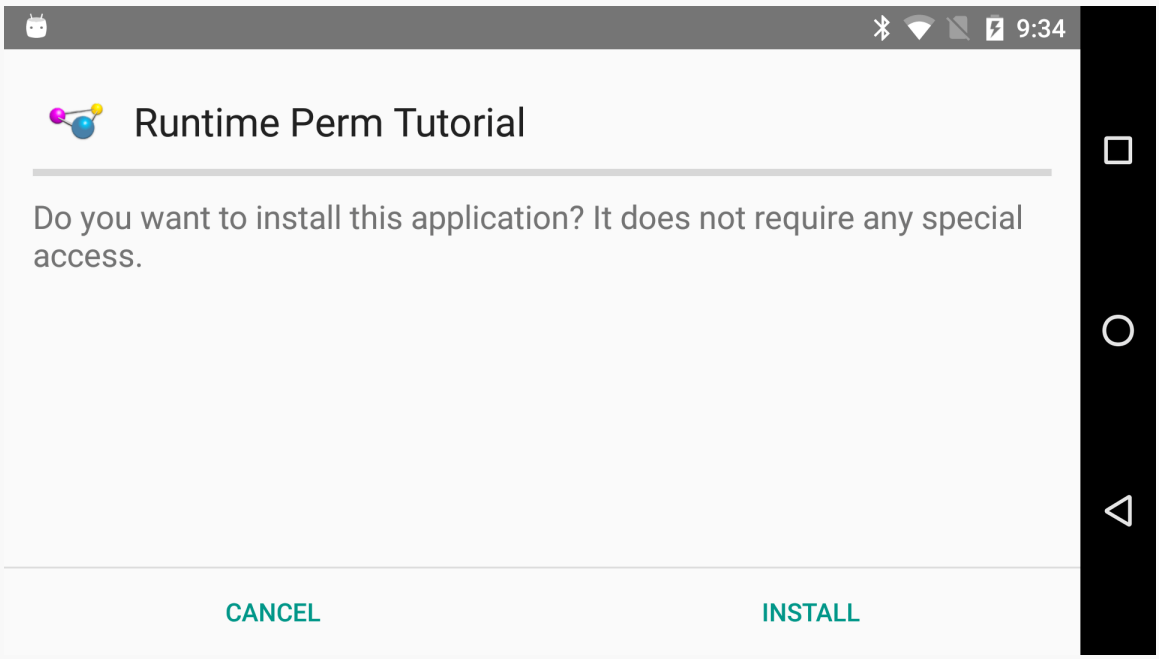


Figure 262: Permission Confirmation Screen, on Android 6.0

Characteristics of Permissions

Several bits of information make up a permission, and some of those affect app developers or users.

Name

We have already seen that permissions have names, and you use them in the `android:name` attribute of the `<uses-permission>` element to identify a permission that you would like your app to hold.

Android framework-defined permissions will begin with `android.permission`. Permissions from libraries or third-party apps will have some other prefix. Make sure that when you create your `<uses-permission>` element that you are using the *fully-qualified* permission name, including `android.permission` or any other prefix.

Also note that Android is case-sensitive, so make sure you use the case of the permission as documented (e.g., `android.permission.INTERNET`). Some versions of Android Studio had a bug where if you let the IDE auto-complete a `<uses-permission>` element for you, sometimes it would have the `android:name`

value appear IN ALL CAPS. This is a bug that has since been fixed, so hopefully it will not affect you in the future.

Protection Level

The definition of a permission, in the framework or in third-party code, will have a "protection level". This describes how the permission itself should be validated. The two protection levels that you will encounter most often are `normal` and `dangerous`.

Normal

A `normal` permission is something that the user *might* care about, but probably not. So, while we need to request the permission in the manifest via `<uses-permission>`, the user will not be bothered about this permission at install time.

The classic example is the `INTERNET` permission. Most Android apps wind up requesting this permission, either for functionality written by the developers or functionality pulled in from libraries (e.g., ad banners). `INTERNET` is considered `normal`, so while we need to request the `INTERNET` permission in the manifest, the user is not informed about this permission anymore at install time.

(the "anymore" note is because in the early days of Android, users were informed about all permissions, regardless of protection level)

Users can see `normal` permissions, though, in other places:

- the list of permissions shown on the Play Store when clicking on a "Permissions" link
- the list of permissions shown in Settings for an app
- third-party tools that help the user understand what capabilities are available to the apps that the user has installed

Dangerous

A `dangerous` permission is one that the definers of the permission (e.g., Google) wants to ensure that the user is aware of and has agreed to.

Classically, this meant that the user would be prompted for this permission at install time. On old versions of Android and the Play Store, `dangerous` permissions would be listed before `normal` permissions.

With Android 6.0+, while `dangerous` permissions are not displayed at install time (for apps with a `targetSdkVersion` of 23 or higher), they will be displayed to the user while the app is running, before the app tries doing something that requires one of those permissions. This is a significant behavior change, so we will be covering it in depth later in this chapter.

Permission Group

Permissions are collected into permission groups.

In the early days of Android, app developers were oblivious to this, as permission groups had no effect on app development, runtime behavior, or user experience.

In the past few years, the "permission" prompts at install time have really been prompting about permission groups. The user is told that the app is requesting permissions from certain groups. Moreover, the blessing that the user gives — by virtue of continuing to install the app — is by *group*, not by permission. If some future update to the app would ask for a new permission, but one from a group that the user agreed to previously, the user would not be informed about this new permission request.

With Android 6.0, permission groups also extend to the runtime permission UX, as while we developers will still request individual permissions, the user will be asked to grant rights with respect to permission groups.

Maximum SDK Version

`<uses-permission>` can have an `android:maxSdkVersion` attribute. This indicates the highest API level for which we need the permission. If the app is running on newer versions of Android, skip the permission.

This is for cases where Android relaxes restrictions over time. We will see an example of this, in the form of the `WRITE_EXTERNAL_STORAGE` permission, in an upcoming chapter.

Minimum SDK Version

You might think that `<uses-permission>` would have an `android:minSdkVersion` attribute to serve as the counterpart to `android:maxSdkVersion`. The `minSdkVersion`

would indicate the lowest API level for which to request a permission; older devices would skip the permission.

Alas, this is not available.

However, there is the awkwardly-named `<uses-permission-sdk-23>` element.

This element functions identically to `<uses-permission>` on Android 6.0+ devices. On older devices, it is ignored.

This element illustrates a problem with the permission system in Android: you have to put all permissions that you want in the manifest. Prior to the runtime permission system in Android 6.0, this would mean that developers who need some controversial permission (e.g., `READ_CONTACTS`) for some fringe feature would need to request the permission from *everyone*, not just those who use the feature. As we will see, the runtime permission system lets us not bother the user until they try using the secured feature. `<uses-permission-sdk-23>` would allow us to not bother with the permission at all on older devices, where its presence might scare away potential users.

New Permissions in Old Applications

Sometimes, Android introduces new permissions that govern behavior that formerly did not require permissions. `WRITE_EXTERNAL_STORAGE` is one example – originally, applications could write to external storage without any permission at all. Android 1.6 introduced `WRITE_EXTERNAL_STORAGE`, required before you can write to external storage. However, applications that were written before Android 1.6 could not possibly request that permission, since it did not exist at the time. Breaking those applications would seem to be a harsh price for progress.

What Android does is "grandfather" in certain permissions for applications supporting earlier SDK versions.

For example, if your `minSdkVersion` is 3 or lower, saying that you support Android 1.5, your application will automatically request `WRITE_EXTERNAL_STORAGE` and `READ_PHONE_STATE`, even if you do not explicitly request those permissions. People installing your application on an Android 1.5 device will see these requests.

Eventually, when you drop support for the older version (e.g., switch to `minSdkVersion` of 4 or higher), Android will no longer automatically request those

permissions. Hence, if your code really *does* need those permissions, you will need to ask for them yourself.

Android 6.0+ Runtime Permission System

In Android 6.0 and higher devices, permissions that are considered to be `dangerous` not only have to be requested via `<uses-permission>` elements, but you *also* have to ask the user to grant you those permissions at runtime. What you gain, though, is that users are not bothered with these permissions at install time, and you can elect to delay asking for certain permissions until such time as the user actually does something that needs them.

This section will occasionally point out snippets of code from the `Permissions/PermissionMonger` sample project.

Let's explore the runtime permissions system via a new series of questions.

What Permissions Are Affected By This?

There are nine permission groups that Android 6.0 manages as user-controllable permissions:

Permission Group	Permission
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	GET_ACCOUNTS, READ_CONTACTS, WRITE_CONTACTS
LOCATION	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	ADD_VOICEMAIL, CALL_PHONE, PROCESS_OUTGOING_CALLS, READ_CALL_LOG, READ_PHONE_STATE, USE_SIP, WRITE_CALL_LOG
SENSORS	BODY_SENSORS
SMS	READ_CELL_BROADCASTS, READ_SMS, RECEIVE_SMS, RECEIVE_MMS, RECEIVE_WAP_PUSH, SEND_SMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

Users will be able to revoke permissions by group, through the Settings app. They can go into the page for your app, click on Permissions, and see a list of the permission groups for which you are requesting permissions:

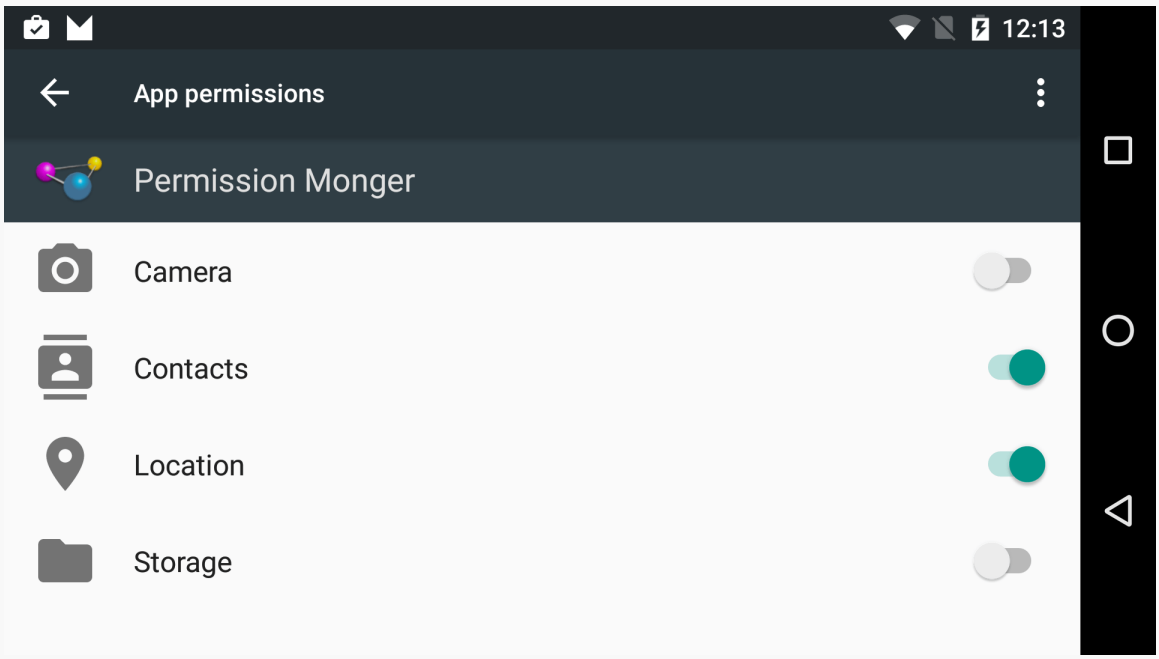


Figure 263: Settings Screen for Permission Monger, Showing Permissions

What Goes in the Manifest?

The same `<uses-permission>` elements as before. These declare the superset of all possible permissions that you can have. If you do not have a `<uses-permission>` element for a particular permission, you cannot ask for it at runtime, and the user cannot grant it to you.

How Do I Know If I Have Permission?

On Android 6.0+, you can call a `checkSelfPermission()` method, available on any `Context` (e.g., your `Activity`). This will return either `PERMISSION_GRANTED` or `PERMISSION_DENIED`, depending on whether or not the user granted you permission or you were automatically given permission (e.g., for `normal` permissions).

For a simpler `boolean` check to see if you have the permission, you could have your own `hasPermission()` method:

```
private boolean hasPermission(String perm) {
  return(PackageManager.PERMISSION_GRANTED==checkSelfPermission(perm));
}
```

(from Permissions/PermissionMonger/app/src/main/java/com/commonsware/android/permmonger/MainActivity.java)

Then you can use that `hasPermission()` call where you need it.

For example, the PermissionMonger app requests five permissions in the manifest:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

(from Permissions/PermissionMonger/app/src/main/AndroidManifest.xml)

The UI is then a table showing the current status of those five permissions:

```
private void updateTable() {
  location.setText(String.valueOf(canAccessLocation()));
  camera.setText(String.valueOf(canAccessCamera()));
  internet.setText(String.valueOf(hasPermission(Manifest.permission.INTERNET)));
  contacts.setText(String.valueOf(canAccessContacts()));
  storage.setText(String.valueOf(hasPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE)));
}

private boolean canAccessLocation() {
  return(hasPermission(Manifest.permission.ACCESS_FINE_LOCATION));
}

private boolean canAccessCamera() {
  return(hasPermission(Manifest.permission.CAMERA));
}

private boolean canAccessContacts() {
  return(hasPermission(Manifest.permission.READ_CONTACTS));
}
```

(from Permissions/PermissionMonger/app/src/main/java/com/commonsware/android/permmonger/MainActivity.java)

At the outset, we only have the one "normal" permission: `INTERNET`:

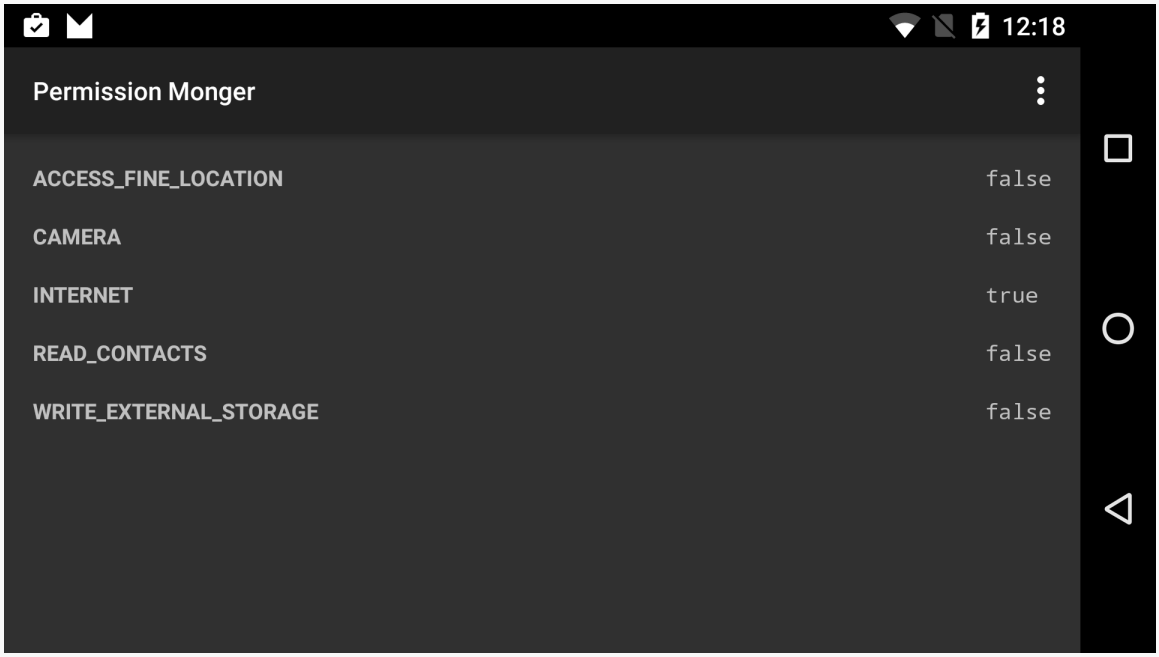


Figure 264: Permission Monger, Showing Initial Permissions

The `checkSelfPermission()` method on `Context` is only available on API Level 23. You can, if you wish, wrap your call to `checkSelfPermission()` in a check of the API level of the device you are running on:

```
if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
  if (checkSelfPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE)==
    PackageManager.PERMISSION_GRANTED) {
    // do something cool
  }
}
```

A simpler approach is to use `ContextCompat`, from the `support-v4` library. This has a `static` implementation of `checkSelfPermission()` that takes a `Context` and your permission string as parameters. It returns the same value (e.g., `PackageManager.PERMISSION_GRANTED`) as does the `checkSelfPermission()` that ships with Android 6.0. But, if you are running on an older device, it checks the version for you and returns `PackageManager.PERMISSION_GRANTED` for older devices. So, the above code snippet turns into:

```
if (ContextCompat.checkSelfPermission(this,
  Manifest.permission.WRITE_EXTERNAL_STORAGE)==
    PackageManager.PERMISSION_GRANTED) {
```

```
  // do something cool
}
```

(assuming that `this` is a subclass of `Context`, like `Activity`)

How Do I Know If the User Takes Permissions Away From Me?

If the user grants you access to some permission group, the only way the user can revoke that is via the Settings app. If the user *does* revoke access to a permission group, your process is terminated.

Hence, while your code is running, you will have all permissions that you started with, plus any new ones that the user grants on the fly based upon your request. There should be no circumstance where your process is running yet you lose a permission.

That being said, your app is not specifically notified about losing the permission. You should be calling `checkSelfPermission()` to determine what you can and cannot do, at least for every process invocation. And, since the call appears to be reasonably cheap, you should just call it whenever you need to know whether you can perform a particular operation.

Note that usually your app will be in the background, if it is running at all, at the time when a runtime permission is revoked. Particularly in an Android 7.0+ multi-window environment, though, it is possible that you will still be visible when the user revokes a runtime permission. Your process is terminated in any case, and if you were visible, your UI is removed from the screen.

How Do I Ask the User For Permission?

To ask the user for one of the runtime permissions, call `requestPermissions()` on your `Activity`. This takes a `String` array of the permissions that you are requesting and a locally-unique integer to identify this request from any other similar requests that you may be making. This `int` serves in much the same role as does the `int` passed into `startActivityForResult()`, though you should keep the value to 8 bits (`0` to `255`) for maximum compatibility.

For example, PermissionMonger will check in `onCreate()` to see if we can access locations or access contacts, and if not, it will request access to those two permissions:

```
    if (!canAccessLocation() || !canAccessContacts()) {
      requestPermissions(INITIAL_PERMS, INITIAL_REQUEST);
    }
```

(from Permissions/PermissionMonger/app/src/main/java/com/commonsware/android/permmonger/MainActivity.java)

`INITIAL_PERMS` and `INITIAL_REQUEST` are just `static final` data members:

```
  private static final String[] INITIAL_PERMS={
    Manifest.permission.ACCESS_FINE_LOCATION,
    Manifest.permission.READ_CONTACTS
  };
```

(from Permissions/PermissionMonger/app/src/main/java/com/commonsware/android/permmonger/MainActivity.java)

```
  private static final int INITIAL_REQUEST=1337;
```

(from Permissions/PermissionMonger/app/src/main/java/com/commonsware/android/permmonger/MainActivity.java)

When the app is first launched, dialogs will appear, one per permission that you requested, asking the user if they would be so kind as to allow your app to do the things that you requested:

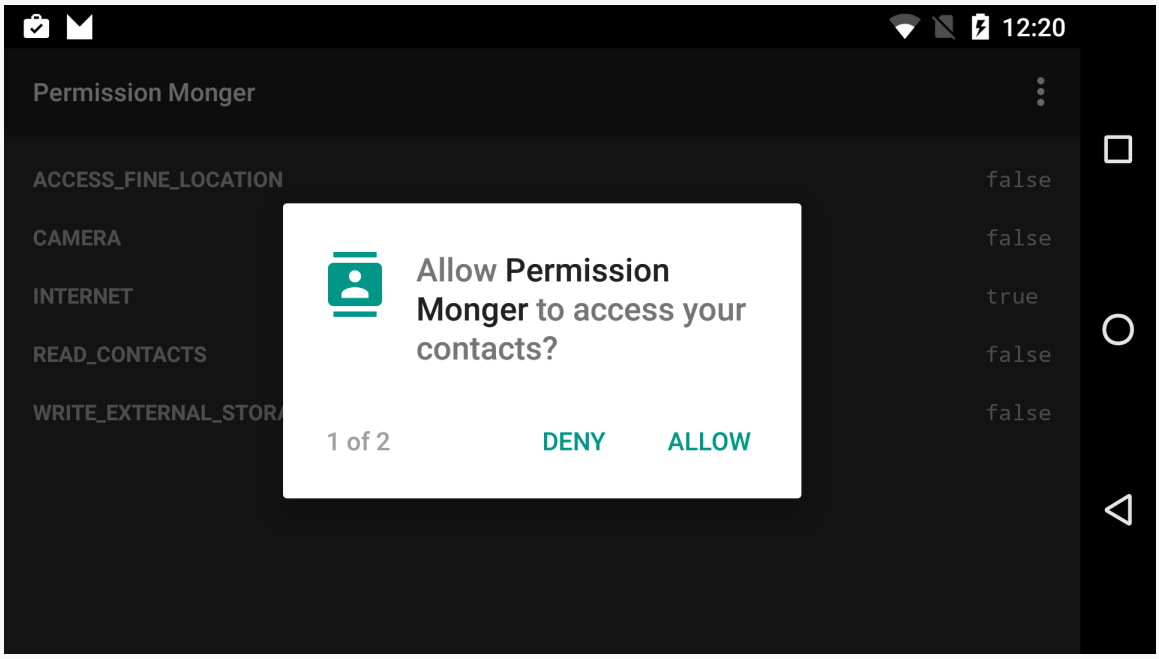


Figure 265: Permission Monger, Requesting READ_CONTACTS Permission

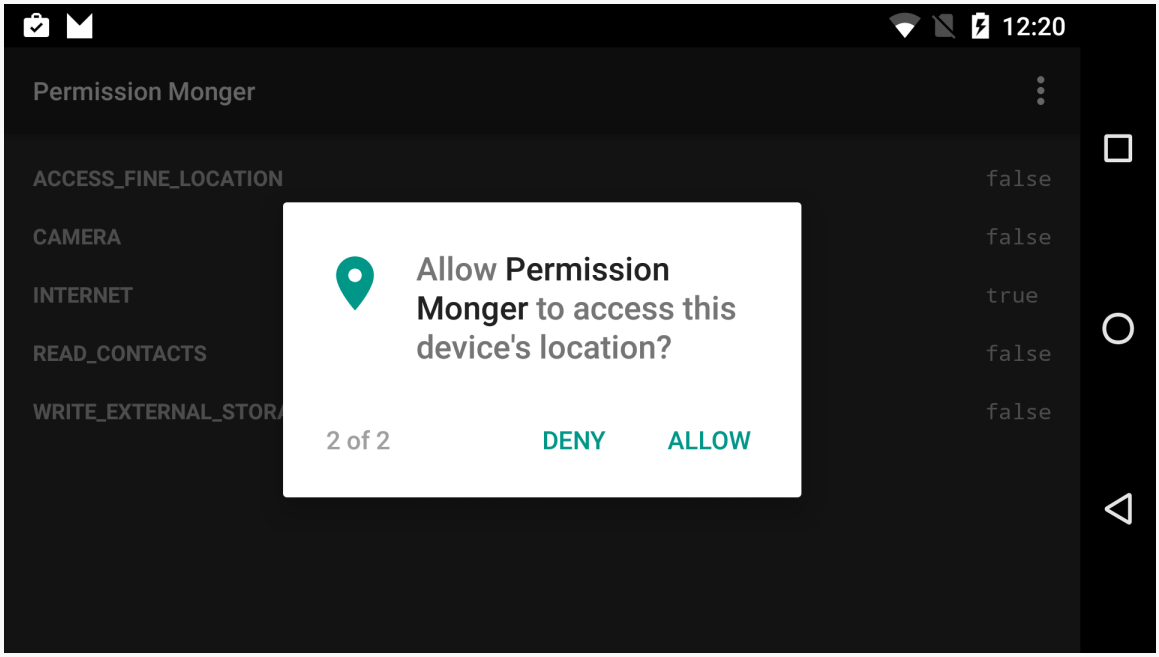


Figure 266: Permission Monger, Requesting ACCESS_FINE_LOCATION Permission

When the user has proceeded through the dialogs, you will be called with `onRequestPermissionsResult()`. You are passed three parameters:

- the locally-unique integer from your `requestPermissions()` call, to identify which `requestPermissions()` call this is the result for
- a `String` array of the requested permissions
- an `int` array of the corresponding results (`PERMISSION_GRANTED` or `PERMISSION_DENIED`)

Whether you use those latter two parameters or simply call `checkSelfPermission()` again is up to you. Regardless, at this point, you should determine what you got, so you know how to react, such as disabling things that the user cannot use given the lack of permission.

Just as `ContextCompat` offers a backwards-compatible implementation of `checkSelfPermission()`, `ActivityCompat` offers a backwards-compatible implementation of `requestPermissions()` that you can use. Otherwise, you will want to take other steps to ensure that you only call `requestPermissions()` on API Level 23+ devices.

When Do I Ask the User For Permission?

That depends a bit on the nature of the permission.

In an ideal world, your app can function without any of the revocable permissions granted to you, albeit perhaps in a limited fashion. In that case, you might ask for permission only when the user tries to do something (e.g., taps on an action bar item) for which you definitely need the permission.

However, sometimes you will need permission to be at all useful to the user. In that case, you will need to ask for permission when the app opens.

In either case, though, bear in mind that while the user will see the dialog asking for permission, the user may not understand *why* you are asking for this permission. You need to make sure that the user understands the cost/benefit trade-off in granting the permission — in other words, what does the user get out of the deal?

For permissions that you are requesting based on user input, you might pop your own dialog or other UI explaining what you want and why you want it, before calling `requestPermissions()`. For permissions that you would want to ask for when the app starts up, make sure that you clearly explain the need for the permissions and what the user gets in exchange as part of a one-time introductory tutorial, one that might also be accessed via an overflow item or nav drawer entry as part of your app's help facility.

When Do I *Not* Ask the User For Permission?

One limitation with the `requestPermissions()` implementation is that it is oblivious to configuration changes.

For example, suppose that in `onCreate()` of your activity, you check to see if you have been granted a runtime permission (via `checkSelfPermission()`), and if you have not, you call `requestPermissions()` to request it from the user. This displays the dialog. Now the user rotates the screen. If the user denies the permission, by default, the user will immediately see the permission dialog again... because your activity will have been destroyed and recreated, and your `onCreate()` will see that you do not have the permission, and so you ask for it again.

In cases like this, you will need to track whether you are in the permission-request flow (e.g., via a `boolean` saved in the instance state `Bundle`) and skip requesting the permission if you have been recreated in the middle of that flow.

What Do I Do If the User Says "No"?

If you were requesting permission as a direct response to some bit of user input (e.g., user tapped on an action bar item), and the user rejects the permission you need to do the work, obviously you cannot do the work. Depending on overall flow, showing a dialog or something to explain why you cannot do what the user asked for may be needed. In some cases, you may deem it to be obvious, by virtue of the fact that the user saw the permission-request dialog and said "deny".

If you were requesting permission pre-emptively, such as when the activity starts, you will need to decide whether that decision needs to be reflected in the current UI (e.g., "no data available" messages, disabled action bar items).

One thing you can do to help here is to detect when this has occurred before you request permissions again. Before you call `requestPermissions()`, you can call `shouldShowRequestPermissionRationale()`, supplying the name of a permission. This will return `true` if the user had previously declined to grant you permission, in cases where Android thinks that the user might benefit from learning a bit more about *why* you need the permission. You can use this to determine whether you should show some explanatory UI of your own first, before continuing with the permission request, or if you should just go ahead and call `requestPermissions()`.

Note that `ActivityCompat` also has a backwards-compatible implementation of `shouldShowRequestPermissionRationale()`, so you can avoid your own API level checks.

What Do I Do If the User Says "No, And Please Stop Asking"?

The second time you ask a user for a particular runtime permission, the user will have a "Never ask again" checkbox:

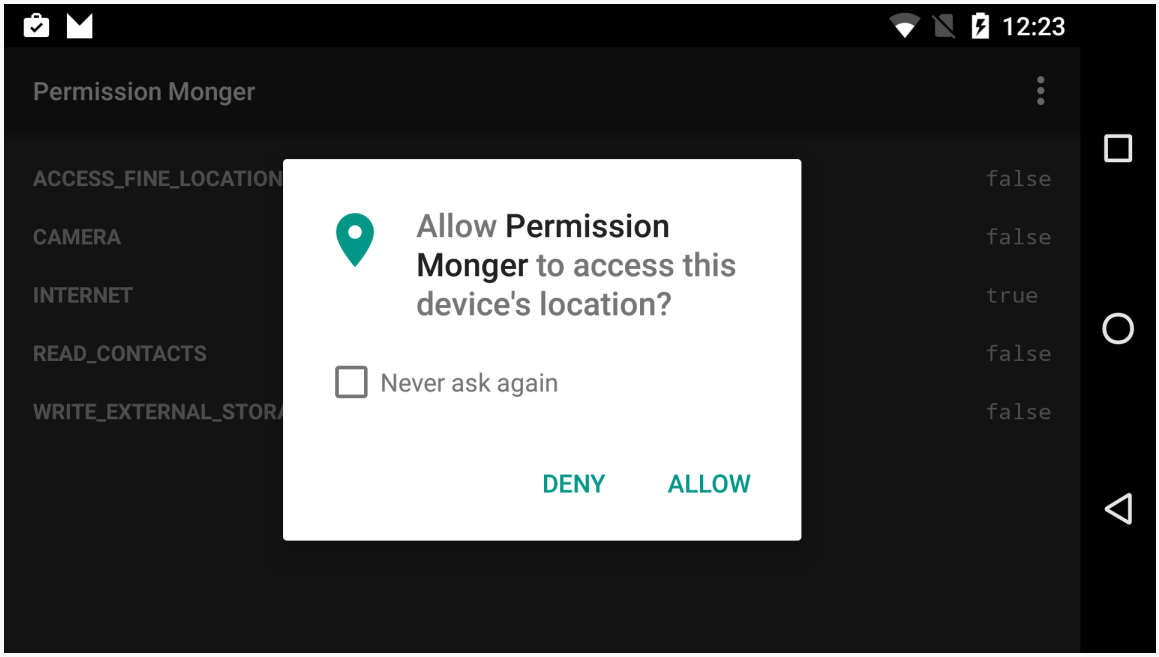


Figure 267: Permission Monger, Requesting ACCESS_FINE_LOCATION Permission (Again)

If the user checks that and clicks the Deny button, not only will you not get the runtime permission now, but all future requests will immediately call `onRequestPermissionsResult()` indicating that your request for permission was denied. The only way the user can now grant you this permission is via the Settings app.

You need to handle this situation with grace and aplomb.

Choices include:

- Disabling UI input (e.g., action bar items) that cannot be performed because you lack permission
- Display a dialog, explaining the situation, with a button that links the user over to your app's screen in Settings, so the user can grant you this permission
- Displaying inline messages about why you cannot show data (e.g., a list of contacts that you cannot show because the user did not grant you access), perhaps with a hyperlink that displays a screen with additional information about the situation

For permissions that, when denied, leave your app in a completely useless state, you may wind up just displaying a screen on app startup that says "sorry, but this app is useless to you", with options for the user to uninstall the app or grant you the desired permissions.

Note that `shouldShowRequestPermissionRationale()` returns `false` if the user declined the permission and checked the checkbox to ask you to stop pestering the user.

What Happens When I Ship This to an Older Device?

Older devices behave as they always have. Since you still list the permissions in the manifest, those permissions will be granted to you if the user installs the app, and the user will be notified about those permissions as part of the installation process. If you are checking the API level yourself, or you are using `ContextCompat` and `ActivityCompat` as described above, your code should just work.

What Happens When My App Has a Lower Target SDK Version?

Apps with a `targetSdkVersion` below 23, on the surface, behave on Android 6.0+ as they would on an older device: the user is prompted for all permissions, and the app is granted those permissions if the app is installed.

However, the user will *still* be able to go into Settings and revoke permissions from these apps, for any permissions the app requests that are in one of the runtime permission groups.

Generally, you will wind up ignoring the issue. All your calls to methods protected by permissions that the user revoke will still "work", insofar as they will not throw a `SecurityException`. However, you just will not get any results back or have the intended effects. So, for example, if you try to `query()` the `ContactsContract` `ContentProvider`, and the user revoked your access to contact-related permissions, the `query()` will return an empty `Cursor`. This is a completely valid response, even ignoring the permission issue, as it is entirely possible that the user has no contacts. Your app should be handling these cases gracefully anyway. Hence, in theory, even if you do nothing special regarding the lost permissions, your app should survive, albeit with reduced functionality for the user. Dave Smith outlines the expected results for legacy apps calling methods sans permission.

However, all else being equal, you should set your `targetSdkVersion` to at least 23 and opt into the runtime permission system.

What Happens if the User Clears My App's Data?

If the user clears your app's data through the Settings app, the runtime permissions are cleared as well. Behavior at this point will be as if your app had been just installed — `checkSelfPermission()` will return `PERMISSION_DENIED`, and you will need to request the permissions.

How Can I Automate Permission Grants?

While the runtime permission system provides a reasonable user-facing UI, having to deal with that UI constantly as a developer can be a significant pain. For testing and debugging purposes, there are some command-line options for granting and revoking permissions that you can use.

Should I Be Using PermissionChecker?

`checkSelfPermission()` on `ContextCompat` always returns `PERMISSION_GRANTED` if either:

- Your app has a `targetSdkVersion` below 23, or
- Your app is running on a device older than Android 6.0

`PERMISSION_DENIED` will be returned only if you have opted into the new runtime permission system (`targetSdkVersion` of 23 or higher), you are running on Android 6.0 or higher, and the user either never granted the permission or revoked it through the Settings app.

The key is that even if you are running on Android 6.0, with an older `targetSdkVersion` (so all permissions are requested at install time and are granted to you automatically), `checkSelfPermission()` still returns `PERMISSION_GRANTED` even if the user revoked the permission in Settings.

The Android Support libraries — specifically `support-v4` — added a `PermissionChecker` class with a `checkPermission()` static method. If you are running on Android 6.0+ with an older `targetSdkVersion`, `checkPermission()` will return `PERMISSION_DENIED_APP_OP` if the user revoked the permission in Settings.

Hence, `PermissionChecker` is useful in cases where you have a really large code base, and you want to try to better handle cases where users revoke permissions, but you are not in position to do a complete implementation of the runtime permissions

system. However, it is merely a stopgap — your long-term plan should be to raise your `targetSdkVersion` to 23 or higher and implement the runtime permissions properly.

A Simple Runtime Permission Abstraction

Occasionally, our need for runtime permissions is fairly straightforward: we need them to do anything meaningful in the app, and so we need to request them up front and exit the app if the user declines to grant one or more of them. Some of the samples in this book fit this model, in part because many of the samples in this book are fairly small apps.

Some of those apps will use a variation of an `AbstractPermissionActivity`, to hide all the runtime permission work, allowing the sample's "real" activity to focus on demonstrating the portion of the Android SDK that the sample is tied to.

One such `AbstractPermissionActivity` can be found in the `Files/FilesEditor` sample project. Later in the book, we will look at this sample to see how to do file I/O on Android. Here, though, let's take a look at `AbstractPermissionActivity`.

Examining the Protocol

The idea is that `AbstractPermissionActivity` will handle the runtime permissions, in its `onCreate()` implementation. Subclasses will need to implement three methods to make this work:

- `getDesiredPermissions()`, which returns the names of the permissions that the app wants
- `onReady()`, which will be called once permission is granted by the user, and serves as an `onCreate()` substitute for the subclass
- `onPermissionDenied()`, which will be called if the user declines granting the permission, so the subclass can do something (e.g., show a `Toast`, then `finish()` and go away)

Requesting the Permission

`onCreate()` will see if we have the desired permissions, and if not, it will call `requestPermissions()` to ask for those that we do not already hold:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  this.state=savedInstanceState;

  if (state!=null) {
    isInPermission=state.getBoolean(STATE_IN_PERMISSION, false);
  }

  if (hasAllPermissions(getDesiredPermissions())) {
    onReady(state);
  }
  else if (!isInPermission) {
    isInPermission=true;

    ActivityCompat
      .requestPermissions(this,
        netPermissions(getDesiredPermissions()),
        REQUEST_PERMISSION);
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/AbstractPermissionActivity.java)

`hasAllPermissions()` iterates over the permission array from `getDesiredPermissions()` and returns `true` if we hold them all, `false` otherwise:

```
private boolean hasAllPermissions(String[] perms) {
  for (String perm : perms) {
    if (!hasPermission(perm)) {
      return(false);
    }
  }

  return(true);
}

protected boolean hasPermission(String perm) {
  return(ContextCompat.checkSelfPermission(this, perm)==
    PackageManager.PERMISSION_GRANTED);
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/AbstractPermissionActivity.java)

If we hold all of the permissions, we go ahead and call `onReady()`, so the activity can start its real work. Otherwise, we call `requestPermissions()` on `ActivityCompat`,

using a `netPermissions()` method to identify those permissions that we do not already hold:

```
private String[] netPermissions(String[] wanted) {
  ArrayList<String> result=new ArrayList<String>();

  for (String perm : wanted) {
    if (!hasPermission(perm)) {
      result.add(perm);
    }
  }

  return(result.toArray(new String[result.size()]));
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/AbstractPermissionActivity.java)

Handling the Result

In `onRequestPermissionResult()`, depending on whether we now hold all the desired permissions, we call `onReady()` or `onPermissionDenied()`:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  isInPermission=false;

  if (requestCode==REQUEST_PERMISSION) {
    if (hasAllPermissions(getDesiredPermissions())) {
      onReady(state);
    }
    else {
      onPermissionDenied();
    }
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/AbstractPermissionActivity.java)

Dealing with (Configuration) Change

It is possible that the user will rotate the device or otherwise trigger a configuration change while our permission dialog is in the foreground. Since our activity is still

visible behind that dialog, we get destroyed and recreated... but we do not want to re-raise the permission dialog again.

That is why we have a `boolean`, named `isInPermission`, that tracks whether or not we are in the middle of requesting permissions. We hold onto that value in `onSaveInstanceState()`:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putBoolean(STATE_IN_PERMISSION, isInPermission);
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/AbstractPermissionActivity.java)

We restore it in `onCreate()`. If we do not hold all of the desired permissions, but `isInPermission` is `true`, we skip requesting the permissions, since we are in the middle of doing so already.

As noted earlier, later in the book, we will look at how a subclass of `AbstractPermissionActivity` works, and you will see `AbstractPermissionActivity` used in other sample apps as well.

Assets, Files, and Data Parsing

Android offers a few structured ways to store data, notably `SharedPreferences` and local SQLite databases. And, of course, you are welcome to store your data "in the cloud" by using an Internet-based service. We will get to all of those topics shortly.

Beyond that, though, Android allows you to work with plain old ordinary files, either ones baked into your app ("assets") or ones on so-called internal or external storage.

To make those files work — and to consume data off of the Internet — you will likely need to employ a parser. Android ships with several choices for XML and JSON parsing, in addition to third-party libraries you can attempt to use.

This chapter focuses on assets, files, and parsers.

Packaging Files with Your App

Let's suppose you have some static data you want to ship with the application, such as a list of words for a spell-checker. Somehow, you need to bundle that data with the application, in a way you can get at it from Java code later on, or possibly in a way you can pass to another component (e.g., `WebView` for bundled HTML files).

There are three main options here: raw resources, XML resources, and assets.

Raw Resources

One way to deploy a file like a spell-check catalog is to put the file in the `res/raw` directory, so it gets put in the Android application `.apk` file as part of the packaging process as a raw resource.

To access this file, you need to get yourself a `Resources` object. From an activity, that is as simple as calling `getResources()`. A `Resources` object offers `openRawResource()` to get an `InputStream` on the file you specify. Rather than a path, `openRawResource()` expects an integer identifier for the file as packaged. This works just like accessing widgets via `findViewById()` – if you put a file named `words.xml` in `res/raw`, the identifier is accessible in Java as `R.raw.words`.

Since you can only get an `InputStream`, you have no means of modifying this file. Hence, it is really only useful for static reference data. Moreover, since it is unchanging until the user installs an updated version of your application package, either the reference data has to be valid for the foreseeable future, or you will need to provide some means of updating the data. The simplest way to handle that is to use the reference data to bootstrap some other modifiable form of storage (e.g., a database), but this makes for two copies of the data in storage. An alternative is to keep the reference data as-is but keep modifications in a file or database, and merge them together when you need a complete picture of the information. For example, if your application ships a file of URLs, you could have a second file that tracks URLs added by the user or reference URLs that were deleted by the user.

XML Resources

If, however, your file is in an XML format, you are better served not putting it in `res/raw/`, but rather in `res/xml/`. This is a directory for XML resources – resources known to be in XML format, but without any assumptions about what that XML represents.

To access that XML, you once again get a `Resources` object by calling `getResources()` on your `Activity` or other `Context`. Then, call `getXml()` on the `Resources` object, supplying the ID value of your XML resource (e.g., `R.xml.words`). This will return an `XmlResourceParser`, which implements the `XmlPullParser` interface. We will discuss how to use this parser, and the performance advantage of using XML resources, later in this chapter.

As with raw resources, XML resources are read-only at runtime.

Assets

Your third option is to package the data in the form of an asset. You can create an `assets/` directory in your source set (e.g., `src/main/assets`), then place whatever files you want in there. Those are accessible at runtime by calling `getAssets()` on

your `Activity` or other `Context`, then calling `open()` with the path to the file (e.g., `assets/foo/index.html` would be retrieved via `open("foo/index.html")`). As with raw resources, this returns an `InputStream` on the file's contents. And, as with all types of resources, assets are read-only at runtime.

One benefit of using assets over raw resources is the `file:///android_asset/ Uri` prefix. You can use this to load an asset into a `WebView`. For example, for an asset located in `assets/foo/index.html` within your project, calling `loadUrl("file:///android_asset/foo/index.html")` will load that HTML into the `WebView`.

Note that assets are compressed when the APK is packaged. Unfortunately, on Android 1.x/2.x, this compression mechanism has a 1MB file size limit. If you wish to package an asset that is bigger than 1MB, you either need to give it a file extension that will not be compressed (e.g., `.mp3`) or actually store a ZIP file of the asset (to avoid the automatic compression) and decompress it yourself at runtime, using the standard `java.util.zip` classes. This restriction was lifted with Android 3.0, and so if your `minSdkVersion` is 11 or higher, this will not be an issue for you.

Files and Android

On the whole, Android just uses normal Java file I/O for local files. You will use the same `File` and `InputStream` and `OutputWriter` and other classes that you have used time and again in your prior Java development work.

What is distinctive in Android is *where* you read and write. Akin to writing a Java Web app, you do not have read and write access to arbitrary locations. Instead, there are only a handful of directories to which you have any access, particularly when running on production hardware.

Internal vs. External

Internal storage refers to your application's portion of the on-board, always-available flash storage.

External storage refers to storage space that can be mounted by the user as a drive in Windows (or, possibly with some difficulty, as a volume in macOS or Linux). Typically, this too is part of the on-board flash, though in a separate area than what is used for internal storage. Making things more complicated is that the *user* will tend to see this labeled as "Internal Storage" (e.g., in Settings); from the Android

SDK's standpoint, what the user thinks is "Internal Storage" is what the SDK calls external storage.

Standard vs. Cache

On both internal and external storage, you have the option of saving files as a cache, or on a more permanent basis. Files located in a cache directory may be deleted by the OS or third-party apps to free up storage space for the user. Files located outside of cache will remain unless manually deleted.

Yours vs. Somebody Else's

Internal storage is on a per-application basis. Files you write to in your own internal storage cannot be read or written to by other applications... normally. Users who "root" their phones can run apps with superuser privileges and be able to access your internal storage. Most users do not root their phones, and so only your app will be able to access your internal storage files.

Files on external storage, though, are visible to all applications and the user. Anyone can read anything stored there, and any application that requests to can write or delete anything it wants.

Working with Internal Storage

You have a few options for manipulating the contents of your app's portion of internal storage.

One possibility is to use `openFileInput()` and `openFileOutput()` on your `Activity` or other `Context` to get an `InputStream` and `OutputStream`, respectively. However, these methods do not accept file paths (e.g., `path/to/file.txt`), just simple filenames.

If you want to have a bit more flexibility, `getFilesDir()` and `getCacheDir()` return a `File` object pointing to the roots of your files and cache locations on internal storage, respectively. Given the `File`, you can create files and subdirectories as you see fit.

To see how this works, take a peek at the `Files/FilesEditor` sample project.

This application implements a tabbed editor, using a `ViewPager` and a third-party tab library. Each tab is an `EditorFragment`, implementing a large `EditText` widget, akin to what we saw as examples back in the chapter on `ViewPager`.

However, those `ViewPager` samples had no persistence. Whatever you typed stayed in the fragments but was lost when the process was terminated. `FileEditor` instead will save what you enter into files, one file per tab.

The layout for the activity is reminiscent of the `ViewPager` samples, except that we are using an `io.karim.MaterialTabs` widget for the tabs, instead of something like a `PagerTabStrip`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:app="http://schemas.android.com/apk/res-auto"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">

  <io.karim.MaterialTabs
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="48dp"
    app:mtIndicatorColor="@color/accent"
    app:mtSameWeightTabs="true"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>
</LinearLayout>
```

(from Files/FilesEditor/app/src/main/res/layout/main.xml)

That library, `io.karim:materialtabs`, is one of our dependencies, along with the `support-v13` library for `ViewPager` itself:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'io.karim:materialtabs:2.0.2'
    implementation 'com.android.support:support-v13:25.1.0'
}

android {
```

```
  compileSdkVersion 25
  buildToolsVersion '26.0.2'

  defaultConfig {
      minSdkVersion 15
      targetSdkVersion 25
  }
}
```

(from Files/FilesEditor/app/build.gradle)

Other than some slight tweaks for using a `MaterialTabs` for the tabs, the `MainActivity` is not significantly different than the original `ViewPager` examples. It loads up the layout and populates the `ViewPager` and tabs:

```
  @Override
  protected void onReady(Bundle savedInstanceState) {
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);

    pager.setAdapter(new SampleAdapter(this, getFragmentManager()));

    MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);
  }
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/MainActivity.java)

Where things start to depart more significantly from the original samples comes in `SampleAdapter`. Rather than 10 pages, we limit the number of tabs to 2 or 3 in `getCount()`. Whether we support 2 or 3 pages depends on what version of Android we are running on — we will explore this issue more later in this chapter.

Rather than delegate the page titles to the `EditorFragment`, `getPageTitle()` looks up a string resource value from an array, based on the position, and uses that for the title. And `getItem()`... becomes more complicated:

```
package com.commonsware.android.fileseditor;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.os.Build;
import android.os.Environment;
import android.support.v13.app.FragmentPagerAdapter;
import java.io.File;
```

```
public class SampleAdapter extends FragmentPagerAdapter {
  private static final int[] TITLES={R.string.internal,
      R.string.external, R.string.pub};
  private static final int TAB_INTERNAL=0;
  private static final int TAB_EXTERNAL=1;
  private static final String FILENAME="test.txt";
  private final Context ctxt;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);

    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(3);
  }

  @Override
  public Fragment getItem(int position) {
    File fileToEdit;

    switch(position) {
      case TAB_INTERNAL:
        fileToEdit=new File(ctxt.getFilesDir(), FILENAME);
        break;

      case TAB_EXTERNAL:
        fileToEdit=new File(ctxt.getExternalFilesDir(null), FILENAME);
        break;

      default:
        fileToEdit=
            new File(Environment.
                  getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS),
                FILENAME);
        break;
    }

    return(EditorFragment.newInstance(fileToEdit));
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(TITLES[position]));
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/SampleAdapter.java)

Based on the supplied `position`, we create a `File` object representing where the data resides for our `EditorFragment`. Right now, let's focus on the `TAB_INTERNAL` case, where we use `getFilesDir()` to create a `File` object pointing to a `test.txt` file on our internal storage.

The `newInstance()` factory method on `EditorFragment` now takes the `File` object as input, instead of the `position`. A `File` is `Serializable`, and so we can put a `File` into the arguments `Bundle`:

```
static EditorFragment newInstance(File fileToEdit) {
  EditorFragment frag=new EditorFragment();
  Bundle args=new Bundle();

  args.putSerializable(KEY_FILE, fileToEdit);
  frag.setArguments(args);

  return(frag);
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

In `onCreateView()` of `EditorFragment`, we inflate a layout that contains our large `EditText` widget and retrieve that `EditText` widget:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);

  editor=(EditText)result.findViewById(R.id.editor);

  return(result);
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

In addition to an `editor` field for our `EditText`, `EditorFragment` has two other fields. One is a `LoadTextTask`, an `AsyncTask` subclass that we will use to load text from our file into our `EditText`. The other is `loaded`, a simple `boolean` to see if we have loaded our text yet:

```
private EditText editor;
private LoadTextTask loadTask=null;
private boolean loaded=false;
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

In `onViewCreated()`, if we have not yet loaded the text, we kick off a `LoadTextTask` to do just that, passing in the `File` that we put into the arguments `Bundle`:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  if (!loaded) {
    loadTask=new LoadTextTask();
    loadTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
        (File)getArguments().getSerializable(KEY_FILE));
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

`LoadTextTask`, in `doInBackground()`, goes through a typical Java file I/O read-all-the-lines process to read in a text file, if it exists. The resulting string is poured into the `EditText`. In `onPostExecute()`, it updates the `EditText` with the read-in text, plus clears the `loadTask` field and sets `loaded` to `true`:

```
private class LoadTextTask extends AsyncTask<File, Void, String> {
  @Override
  protected String doInBackground(File... files) {
    String result=null;

    if (files[0].exists()) {
      BufferedReader br;

      try {
        br=new BufferedReader(new FileReader(files[0]));

        try {
          StringBuilder sb=new StringBuilder();
          String line=br.readLine();

          while (line!=null) {
            sb.append(line);
            sb.append("\n");
            line=br.readLine();
          }

          result=sb.toString();
        }
        finally {
          br.close();
        }
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception reading file", e);
```

```
      }
    }

    return(result);
  }

  @Override
  protected void onPostExecute(String s) {
    editor.setText(s);
    loadTask=null;
    loaded=true;
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

However, since we are using an `AsyncTask`, we should retain this fragment:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

...and in `onDestroy()`, we should `cancel()` this task if it is still running, as we no longer need the results:

```
@Override
public void onDestroy() {
  if (loadTask!=null) {
    loadTask.cancel(false);
  }

  super.onDestroy();
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

Rather than have some dedicated "save" action bar item or similar UI element, we can just arrange to save the data when our fragment gets paused. This is a typical approach in Android apps, as users do not necessarily get an opportunity to click some "save" UI element, if they get interrupted by a phone call or something. So, in `onPause()`, we kick off a `SaveThread` to write our `EditText` contents to the same `File`, once again pulled from the arguments `Bundle`:

```
@Override
public void onPause() {
  if (loaded) {
    new SaveThread(editor.getText().toString(),
        (File)getArguments().getSerializable(KEY_FILE)).start();
  }

  super.onPause();
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

However, note that we do not fork the `SaveThread` if `loaded` is still `false`. In that case, we know that we are still loading in the text, which means the text cannot possibly have been modified by the user, so there is nothing to save.

`SaveThread` ensures that the directory we want to write to exists (as it may or may not exist, particularly on emulators), then uses Java `Writer` objects to write out our text. Since there is nothing that we want to do with the UI here, a plain `Thread`, rather than an `AsyncTask`, is a better solution:

```
private static class SaveThread extends Thread {
  private final String text;
  private final File fileToEdit;

  SaveThread(String text, File fileToEdit) {
    this.text=text;
    this.fileToEdit=fileToEdit;
  }

  @Override
  public void run() {
    try {
      fileToEdit.getParentFile().mkdirs();

      FileOutputStream fos=new FileOutputStream(fileToEdit);

      Writer w=new BufferedWriter(new OutputStreamWriter(fos));

      try {
        w.write(text);
        w.flush();
        fos.getFD().sync();
      }
      finally {
        w.close();
      }
```

```
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception writing file", e);
    }
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

The reason for using a `FileOutputStream`, and that mysterious `getFD().sync()` part, will be covered later in this chapter.

The result is a set of tabbed editors, where the first one is our one for internal storage:

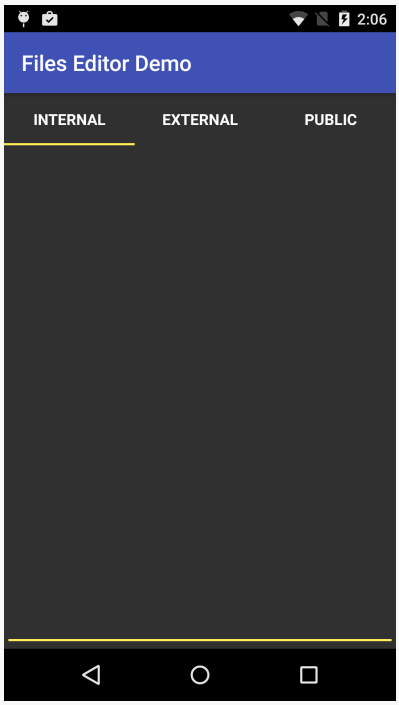


Figure 268: FilesEditor Sample, As Initially Launched

If you type something into the "Internal" tab, press BACK to exit the activity, and go back into the app again, whatever you typed in will be re-loaded from disk and will show up in the editor.

The files stored in internal storage are accessible only to your application, by default. Other applications on the device have no rights to read, let alone write, to this space.

However, bear in mind that some users "root" their Android phones, gaining superuser access. These users will be able to read and write whatever files they wish. As a result, please consider application-local files to be secure against malware but not necessarily secure against interested users.

Working with External Storage

Like internal storage, external storage can be used with ordinary Java file I/O (e.g., the `File` class). What differs is where you can read or write. And, compared to internal storage, what differs is whether you are *allowed* to read or write.

Where to Write

If you have files that are tied to your application that are simply too big to risk putting in internal storage, or if the user should be able to download the files off their device at will, you can use `getExternalFilesDir()`, available on any activity or other `Context`. This will give you a `File` object pointing to an automatically-created directory on external storage, unique for your application. While not secure against other applications, it does have one big advantage: when your application is uninstalled, these files are automatically deleted, just like the ones in the application-local file area. This method was added in API Level 8. This method takes one parameter — typically `null` — that indicates a particular type of file you are trying to save (or, later, load).

In `SampleAdapter` of the sample app, if the user chooses the "External" tab, we use `getExternalFilesDir()` to create the `File` to be used by the `EditorFragment`:

```
      case TAB_EXTERNAL:
        fileToEdit=new File(ctxt.getExternalFilesDir(null), FILENAME);
        break;
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/SampleAdapter.java)

There is also `getExternalCacheDir()`, which returns a `File` pointing at a directory that contains files that you would like to have, but if Android or a third-party app clears the cache, your app will continue to function normally.

If you have files that belong more to the user than to your app — pictures taken by the camera, downloaded MP3 files, etc. — a better solution is to use `getExternalStoragePublicDirectory()`, available on the `Environment` class. This will give you a `File` object pointing to a directory set aside for a certain type of file,

based on the type you pass into `getExternalStoragePublicDirectory()`. For example, you can ask for `DIRECTORY_MOVIES`, `DIRECTORY_MUSIC`, or `DIRECTORY_PICTURES` for storing MP4, MP3, or JPEG files, respectively. These files will be left behind when your application is uninstalled. This method was also added in API Level 8.

In `SampleAdapter` of the sample app, if the user chooses the "Public" tab, we use `getExternalStoragePublicDirectory()` to create the `File` to be used by the `EditorFragment`, putting our file in the `DIRECTORY_DOCUMENTS` location:

```
    default:
      fileToEdit=
          new File(Environment.
                getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS),
              FILENAME);
      break;
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/SampleAdapter.java)

You will also find a `getExternalStorageDirectory()` method on `Environment`, pointing to the root of the external storage. This is no longer the preferred approach — the methods described above help keep the user's files better organized. However, if you are supporting older Android devices, you may need to use `getExternalStorageDirectory()`, simply because the newer options may not be available to you.

Relevant Permissions

On all relevant Android versions prior to Android 4.4 (API Level 19), if you want to write to external storage, you need to hold the `WRITE_EXTERNAL_STORAGE` permission. And, on those versions, you do not need a permission to read from external storage.

On Android 4.4 and up, the rules are a bit different:

- To read or write in the directory trees rooted at `getExternalFilesDir()` and `getExternalCacheDir()`, you do not need a permission
- To write to anywhere else on external storage, you need `WRITE_EXTERNAL_STORAGE`
- To read from anywhere else on external storage, you need either `WRITE_EXTERNAL_STORAGE` (if you already have that) or `READ_EXTERNAL_STORAGE` (if not)

Hence, so long as your `android:minSdkVersion` is less than 19, you need to take the most conservative approach:

- If you are writing *anywhere* on external storage, request the `WRITE_EXTERNAL_STORAGE` permission
- If you are only reading, but from *anywhere* on external storage, request the `READ_EXTERNAL_STORAGE` permission

Note that you might get paths to external storage locations from third-party apps, typically in the form of a `Uri`. If you are handling `Uri` values from third-party apps, you should request `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`, in case the third-party app hands you a `Uri` pointing to external storage.

For example, here is the sample app's manifest, complete with the `<uses-permission>` element for `WRITE_EXTERNAL_STORAGE`:

```
<?xml version="1.0"?>
<manifest package="com.commonsware.android.fileseditor"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
```

```
</manifest>
```

(from Files/FilesEditor/app/src/main/AndroidManifest.xml)

However, on Android 6.0+, `WRITE_EXTERNAL_STORAGE` is one of those `dangerous` permissions that we have to request at runtime. That is why this sample app uses the `AbstractPermissionActivity` profiled in the material on runtime permissions. Overall, our `MainActivity` looks like this:

```
package com.commonsware.android.fileseditor;

import android.os.Bundle;
import android.support.v4.view.ViewPager;
import android.widget.Toast;
import io.karim.MaterialTabs;
import static android.Manifest.permission.WRITE_EXTERNAL_STORAGE;

public class MainActivity extends AbstractPermissionActivity  {
  @Override
  protected String[] getDesiredPermissions() {
    return(new String[]{WRITE_EXTERNAL_STORAGE});
  }

  @Override
  protected void onPermissionDenied() {
    Toast
      .makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG)
      .show();
    finish();
  }

  @Override
  protected void onReady(Bundle savedInstanceState) {
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);

    pager.setAdapter(new SampleAdapter(this, getFragmentManager()));

    MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/MainActivity.java)

`getDesiredPermissions()` indicates that we want `WRITE_EXTERNAL_STORAGE`, and `onPermissionDenied()` exits the app after showing a `Toast`. `onReady()` is where we set up the tabs, as we now have all the permissions that we need to be able to work with external storage.

Note that we do *not* need `WRITE_EXTERNAL_STORAGE` for `getExternalFilesDir()` on API Level 19+ devices. This leads to another possible permission strategy for this app:

- We could add `android:maxSdkVersion="18"` to the `<uses-permission>` element for `WRITE_EXTERNAL_STORAGE`. This would indicate that we only want this permission on devices that are running API Level 18 or lower.
- We could then have `SampleAdapter` see what version of Android we are running on. If we are running on API Level 19 or higher, we know that we did not request `WRITE_EXTERNAL_STORAGE`, but that we do not need that permission for `getExternalFilesDir()`. In that case, we could suppress the "Public" tab (since we do not have permission to write there) and only show two tabs. But, on older devices where we *did* ask for that permission, we could show all three tabs (since we have rights for all of external storage).

When to Write

Also, external storage may be tied up by the user having mounted it as a USB storage device. You can use `getExternalStorageState()` (a static method on `Environment`) to determine if external storage is presently available or not. On Android 3.0 and higher, this should be much less of an issue, as they changed how the external storage is used by the host PC — originally, this used USB Mass Storage Mode (think thumb drives) and now uses the USB Media Transfer Protocol (think MP3 players). With MTP, both the Android device and the PC it is connected to can have access to the files simultaneously; Mass Storage Mode would only allow the host PC to have access to the files if external storage is mounted.

Nowadays, you can use `getStorageState()` on the `EnvironmentCompat` class from the `support-v4` library to find out the state of external storage, for the particular `File` passed as a parameter.

Letting the User See Your Files

The switch to MTP has one side-effect for Android developers: files you write to external storage may not be automatically visible to the user. At the time of this

writing, the only files that will show up on the user's PC will be ones that have been indexed by the `MediaStore`. While the `MediaStore` is typically thought of as only indexing "media" (images, audio files, video files, etc.), it was given the added role in Android 3.0 of maintaining an index of *all* files for the purposes of MTP.

Your file that you place on external storage will not be indexed automatically simply by creating it and writing to it. Eventually, it will be indexed, though it may be quite some time for an automatic indexing pass to take place.

To force Android to index your file, you can use `scanFile()` on `MediaScannerConnection`:

```
String[] paths={pathToYourNewFileOnExternalStorage};
MediaScannerConnection.scanFile(this, paths, null, null);
```

The third parameter to `scanFile()` is an array of MIME types, to line up with the array of paths in the second parameter. If your file is some form of media, and you know the MIME type, supplying that will ensure that your media will be visible as appropriate to the right apps (e.g., images in the Gallery app). Otherwise, Android will try to infer a MIME type from the file extension.

In the sample app, since the `EditorFragment` does not know whether the file is on external storage and therefore is reachable, it does not know whether or not this sort of indexing is appropriate. In a more conventional scenario, where the `EditorFragment` would consistently be writing to external storage, `SaveThread` could arrange to invoke `MediaScannerConnection` as part of its work. However, `scanFile()` needs a `Context`, and so the `SaveThread` would need one of those. You would wind up with something a bit like:

```
  private static class SaveThread extends Thread {
    private final String text;
    private final File fileToEdit;
    private final Context ctxt;

    SaveThread(Context ctxt, String text, File fileToEdit) {
      this.ctxt=ctxt.getApplicationContext();
      this.text=text;
      this.fileToEdit=fileToEdit;
    }

    @Override
    public void run() {
      try {
```

```
      fileToEdit.getParentFile().mkdirs();

      FileOutputStream fos=new FileOutputStream(fileToEdit);

      Writer w=new BufferedWriter(new OutputStreamWriter(fos));

      try {
        w.write(text);
        w.flush();
        fos.getFD().sync();
      }
      finally {
        w.close();
        String[] paths={fileToEdit.getAbsolutePath()};
        MediaScannerConnection.scanFile(ctxt, paths, null, null);
      }
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception writing file", e);
    }
  }
}
```

Here, we use `getApplicationContext()`, which returns to us a `Context` that is a process-wide singleton. That way, if our activity is destroyed while the thread is still running, we still have a valid `Context` to use.

Limits on External Storage Open Files

Many Android devices will have a per-process limit of 1024 open files, on any sort of storage. This is usually not a problem for developers.

On some devices — including probably all that are running Android 4.2 and higher — there is a *global* limit of 1024 open files on external storage. In other words, all running apps combined can only open 1024 files simultaneously on external storage.

This means that it is important for you to minimize how many open files on external storage you have at a time. Having a few open files is perfectly reasonable; having a few hundred open files is not.

Removable Storage

Some Android devices support micro SD card slots, where cards inserted in there are not part of internal or external storage. Some Android devices support USB On-The-Go (OTG) drives. Some Android devices support other forms of removable storage, such as full-size SD cards, full-size USB thumb drives, etc.

And, until Android 4.4, none of that was officially available to you as a developer.

What You Can Do

Android 4.4 (API Level 19) added two new methods, `getExternalCacheDirs()` and `getExternalFilesDirs()`, the plural versions of the classic methods. These return an array of `File` objects, representing one *or more* places where your app can work with external storage. The first element in the array will be the same `File` object returned by the singular versions of the methods (e.g., `getExternalFilesDir()`). The other elements in the array, if any, will represent app-specific directories on removable storage location. The Android Support package has a `ContextCompat` class containing static versions of `getExternalCacheDirs()` and `getExternalFilesDirs()`, so you can use the same code on API Level 4 and above, though the backport will only ever return one directory in the array.

What You Can't Do

You cannot access arbitrary files on removable storage. You have full read/write access to the specific locations referred to by the aforementioned methods, but that is all that you are guaranteed access to. So, you cannot write code that iterates over all the files on removable storage, for example.

The Workarounds

The Storage Access Framework allows you to work with content on removable storage, and in cloud storage providers, though not using `File`.

The `MediaStore` contains an index of all files on external storage and removable storage. You can query for content from certain MIME types (e.g., all videos) and be able to read in that content. However, once again, you are not using `File`.

Multiple User Accounts

On Android 4.1 and earlier, each Android device was assumed to be used by just one person.

On Android 4.2+ tablets — and Android 5.0+ phones — it is possible for a device's owner to set up multiple user accounts. Each user gets their own section of internal and external storage for files, databases, `SharedPreferences`, and so forth. From your standpoint, it is as if the users are really on different devices, even though in reality it is all the same hardware.

However, this means that paths to internal and external storage now may vary by user. Hence, it is *very important* for you to use the appropriate methods, outlined in this chapter, for finding locations on internal storage (e.g., `getFilesDir()`) and external storage (e.g., `getExternalFilesDir()`).

Some blog posts, Stack Overflow answers, and the like will show the use of hard-coded paths for these locations (e.g., `/sdcard` or `/mnt/sdcard` for the root of external storage). Hard-coding such paths was never a good idea. And, as of Android 4.2, those paths are simply wrong and will not work.

On Android 4.2+, for the original user of the device, internal storage will wind up in the same location as before, but external storage will use a different path. For the second and subsequent users defined on the device, both internal and external storage will reside in different paths. The various methods, like `getFilesDir()`, will handle this transparently for you.

Note that, at the time of this writing, multiple accounts are not available on the emulators, only on actual tablets. Phones usually will not have multiple-account support, under the premise that tablets are more likely to be shared than are phones.

Linux Filesystems: You Sync, You Win

Android is built atop a Linux kernel and uses Linux filesystems for holding its files. Classically, Android used YAFFS (Yet Another Flash File System), optimized for use on low-power devices for storing data to flash memory.

YAFFS has one big problem: only one process can write to the filesystem at a time. For those of you into filesystems, rather than offering file-level locking, YAFFS has

partition-level locking. This can become a bit of a bottleneck, particularly as Android devices grow in power and start wanting to do more things at the same time like their desktop and notebook brethren.

Android 3.0 switched to ext4, another Linux filesystem aimed more at desktops/notebooks. Your applications will not directly perceive the difference. However, ext4 does a fair bit of buffering, and it can cause problems for applications that do not take this buffering into account. Linux application developers ran headlong into this in 2008-2009, when ext4 started to become popular. Android developers will need to think about it now... for your own file storage.

If you are using SQLite or `SharedPreferences`, you do not need to worry about this problem. Android (and SQLite, in the case of SQLite) handle all the buffering issues for you. If, however, you write your own files, you may wish to contemplate an extra step as you flush your data to disk. Specifically, you need to trigger a Linux system call known as `fsync()`, which tells the filesystem to ensure all buffers are written to disk.

If you are using `java.io.RandomAccessFile` in a synchronous mode, this step is handled for you as well, so you will not need to worry about it. However, Java developers tend to use `FileOutputStream`, which does not trigger an `fsync()`, even when you call `close()` on the stream. Instead, you call `getFD().sync()` on the `FileOutputStream` to trigger the `fsync()`. Note that this may be time-consuming, and so disk writes should be done off the main application thread wherever practical, such as via an `AsyncTask`.

This is why, in `EditorFragment`, our `SaveThread` implementation looks like this:

```
private static class SaveThread extends Thread {
  private final String text;
  private final File fileToEdit;

  SaveThread(String text, File fileToEdit) {
    this.text=text;
    this.fileToEdit=fileToEdit;
  }

  @Override
  public void run() {
    try {
      fileToEdit.getParentFile().mkdirs();

      FileOutputStream fos=new FileOutputStream(fileToEdit);
```

```
      Writer w=new BufferedWriter(new OutputStreamWriter(fos));

      try {
        w.write(text);
        w.flush();
        fos.getFD().sync();
      }
      finally {
        w.close();
      }
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception writing file", e);
    }
  }
}
```

(from Files/FilesEditor/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

While we use a `Writer` to do the writing, it is wrapped around a `FileOutputStream`, so we can get access to the `FileDescriptor` (via `getFD()`) and call `sync()` on it.

StrictMode: Avoiding Janky Code

Users are more likely to like your application if, to them, it feels responsive. Here, by "responsive", we mean that it reacts swiftly and accurately to user operations, like taps and swipes.

Conversely, users are less likely to be happy with you if they perceive that your UI is "janky" — sluggish to respond to their requests. For example, maybe your lists do not scroll as smoothly as they would like, or tapping a button does not yield the immediate results they seek.

While threads and `AsyncTask` and the like can help, it may not always be obvious where you should be applying them. A full-scale performance analysis, using Traceview or similar Android tools, is certainly possible. However, there are a few standard sorts of things that developers do, sometimes quite by accident, on the main application thread that will tend to cause sluggishness:

1. Flash I/O, both for internal and external storage
2. Network I/O

However, even here, it may not be obvious that you are performing these operations on the main application thread. This is particularly true when the operations are really being done by Android's code that you are simply calling.

That is where `StrictMode` comes in. Its mission is to help you determine when you are doing things on the main application thread that might cause a janky user experience.

`StrictMode` works on a set of policies. There are presently two categories of policies: VM policies and thread policies. The former represent bad coding practices that pertain to your entire application, notably leaking SQLite `Cursor` objects and kin. The latter represent things that are bad when performed on the main application thread, notably flash I/O and network I/O.

Each policy dictates what `StrictMode` should watch for (e.g., flash reads are OK but flash writes are not) and how `StrictMode` should react when you violate the rules, such as:

1. Log a message to Logcat
2. Display a dialog
3. Crash your application (seriously!)

The simplest thing to do is call the static `enableDefaults()` method on `StrictMode` from `onCreate()` of your first activity. This will set up normal operation, reporting all violations by simply logging to Logcat. However, you can set your own custom policies via `Builder` objects if you so choose.

However, do not use `StrictMode` in production code. It is designed for use when you are building, testing, and debugging your application. It is not designed to be used in the field.

So, for example, you might have something like this in your launcher activity:

```
StrictMode.ThreadPolicy.Builder b=new StrictMode.ThreadPolicy.Builder();

if (BuildConfig.DEBUG) {
  b.detectAll().penaltyDeath();
}
else {
  b.detectAll().penaltyLog();
}

StrictMode.setThreadPolicy(b.build());
```

`BuildConfig.DEBUG` will be `true` for debuggable builds, false otherwise. So, in the case of a debug build, we want to detect all mistakes and crash the app immediately when we encounter them, but in production, we want to just log information about the mistake to Logcat.

You will note that the sample app does not contain this code. That is because calling methods like `getFilesDir()` and `getExternalFilesDir()` really ought to be on background threads, as `StrictMode` will complain about them. Hence, this code would cause `SampleAdapter` to crash when it tries building the `File` object to use. This could be rectified by having `SampleAdapter` simply pass in a flag indicating the storage location and having `LoadThreadTask` and `SaveThread` deal with the `File` objects.

Note that `StrictMode` will also report leaked open files. For example, if you create a `FileOutputStream` on a `File` and fail to `close()` it later, when the `FileOutputStream` (and related objects) are garbage-collected, `StrictMode` will report to you the fact that you failed to close the stream. This is very useful to help you make sure that you are not leaking open files that may contribute to exhausting the 1,024 open file limit on external storage.

Files, and Your Development Machine

All this reading and writing of data is nice, but for debugging and diagnostic purposes, it is often useful for you to be able to look at the files, other than through your app.

This is somewhat challenging, due to the lack of tools and due to security restrictions in production devices (as compared to emulators).

That being said, the following sections will outline some options that you have to access your app's files independently of your app.

Mounting as a Drive

If you have an actual Android device, when you plug it in via a USB cable, usually you will get external storage available as a drive letter (Windows) or a mounted volume (macOS and Linux). Depending upon the device, manufacturer, and configuration, you might also have access to removable storage this way as well.

In these cases, you can use your development machine's OS to poke around these file locations and look at your files (or anyone else's).

However, there are some wrinkles:

- On Android 6.0+, by default, a USB connection is only used for charging. You need to slide open the notification tray and tap on the `Notification` for the USB connection, to toggle it to share files using MTP.
- Some versions of macOS and Linux will require you to install additional software to view files over MTP.
- If you see a volume name labeled "Internal Storage", that is really external storage, because confusing people is fun, apparently.
- You cannot get to what the Android SDK refers to as internal storage by this means.

Browsing Files Via Android Studio

Android Studio 3.0+ offers a Device File Explorer tool. As the name suggests, it allows you to explore the filesystems on an Android device (or emulator).

This tool is docked by default in the lower-right screen edge. When opened, it displays a fairly typical directory tree structure:

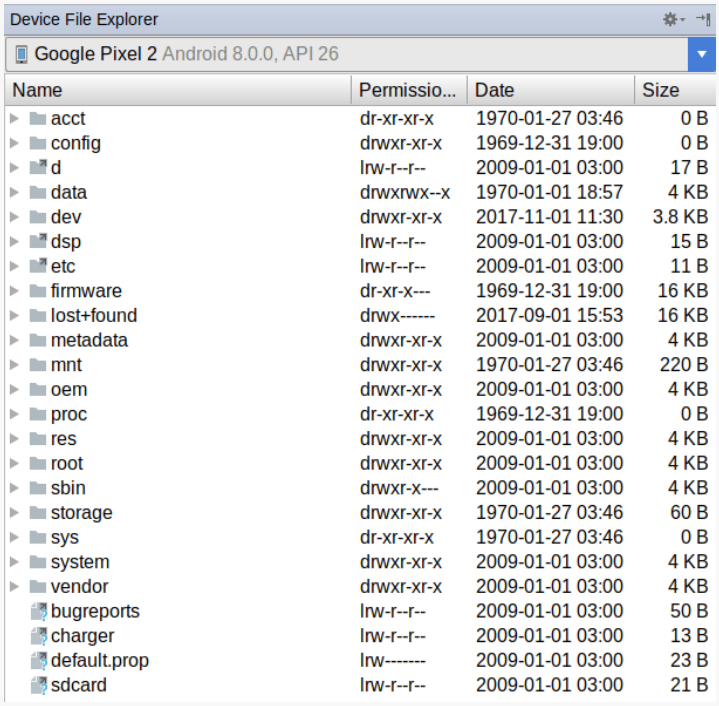


Figure 269: Device File Explorer, As Initially Opened

Key places to try browsing to are:

- `/data/data/.../`, where `...` is an application ID, to browse the internal storage for a particular app for the primary device user
- `/sdcard/` or `/mnt/sdcard/`, which will be typical (though not universal) ways of reaching external storage

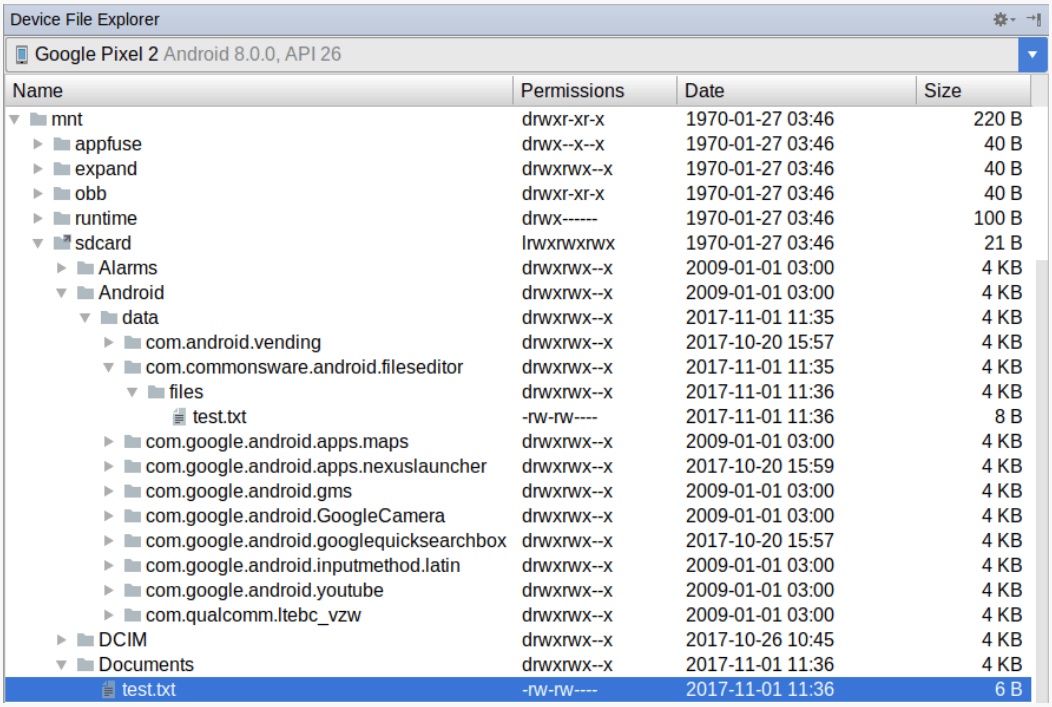


Figure 270: Device File Explorer, Showing External and Public FilesEditor Files

However, for hardware and emulators with the Google proprietary apps (e.g., Play Store), you will only be able to browse into internal storage of debuggable apps. The Explorer has no ability to browse into production apps in these environments:

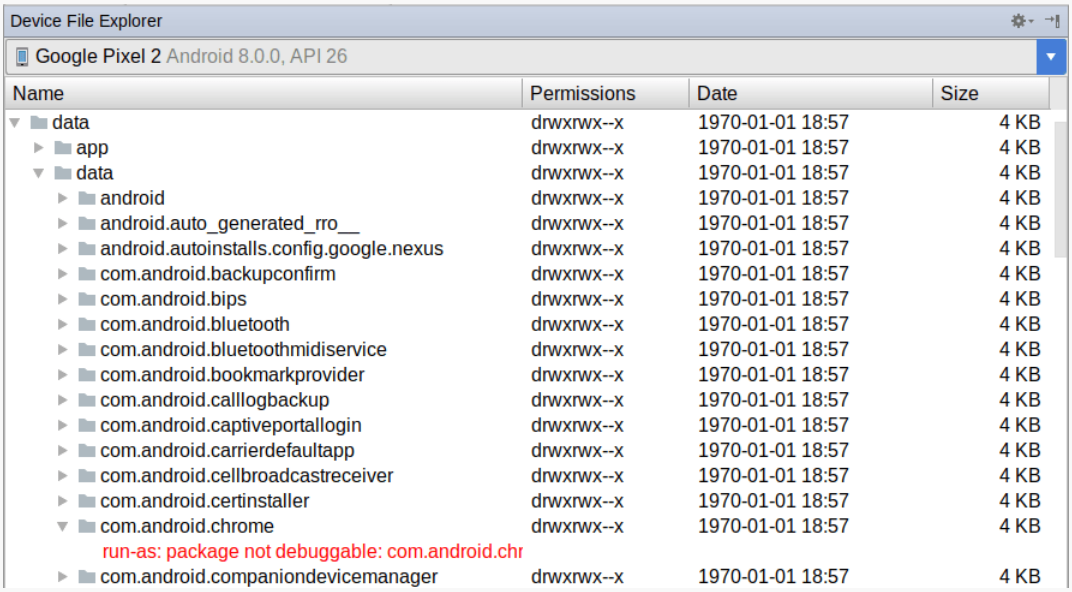


Figure 271: Device File Explorer, Showing Blocked Access to Chrome's Files

On an emulator that lacks the Google proprietary apps, though, you can browse much more of the filesystem, including the internal storage of production apps:

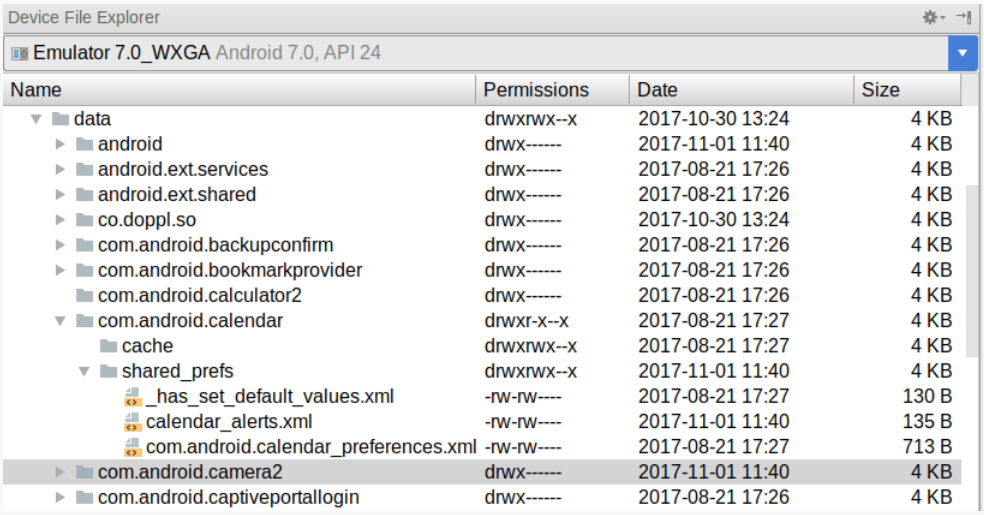


Figure 272: Device File Explorer, Showing Access to Calendar's Files on an Emulator

For any file that you can see, if it is of a recognized file type, double-clicking the file will open it up in Android Studio. That is because, at least on macOS and Linux, a virtual filesystem is added on your development machine that maps to the device or

emulator. You can find the location by looking at the "breadcrumbs" above the editor that show the segments of its local file path:

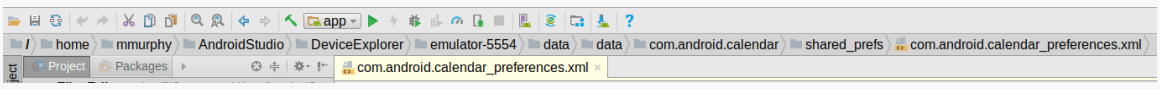


Figure 273: Breadcrumbs Showing Local Map to Emulator Filesystem

You can use that same path to work with the file using tools outside of Android Studio. However, changes that you make to the file through Android Studio will not be saved back to the device or emulator.

Right-clicking over a file will give you options to:

- Open it in Android Studio
- Save it to an arbitrary location on your development machine
- Delete it
- "Synchronize" it, which updates the UI from the filesystem
- Copy its on-device/on-emulator path to the clipboard, for pasting into Android app development books like this one:

```
/data/data/com.android.calendar/shared_prefs/com.android.calendar_preferences.xml
```

Right-clicking over a directory will give you options to:

- Create a new file or subdirectory in this directory
- Save the *contents* of this directory — though not the directory name itself — into a directory of your choice
- Upload a file or directory into this directory
- Delete the directory (recursively deleting all of its contents)
- "Synchronize" it, which updates the UI from the filesystem
- Copy its on-device/on-emulator path to the clipboard

Note that the Device File Explorer effectively shows a snapshot of what is on the device or emulator filesystem. It will only reflect changes if you use the Synchronize context menu option.

Push and Pull for External Storage

You can get at external storage of devices and emulators via the command-line `adb` tool. This program is in `platform-tools/` of your Android SDK installation, and it is

a good idea to add that directory to your operating system's PATH environment variable, so you can run adb from anywhere.

adb push and adb pull allow you to upload and download files, respectively. Both take the local path and the remote (device/emulator) path as command-line arguments, although in varying order:

- adb push localpath remotepath will upload the file represented by localpath to the location represented by remotepath
- adb pull remotepath localpath will download the file represented by remotepath to the location represented by localpath

For external storage, the root directory name varies by Android OS version:

- Android 1.x/2.x: use /sdcard/
- Android 4.x/5.x: use /mnt/shell/emulated/0/
- Android 6.0+: use /storage/emulated/0/.

So, for example, the following command would push an index.html file to the getExternalFilesDir() location for the primary device account, for an app whose application is your.package.name.here:

```
adb push index.html /storage/emulated/0/Android/data/your.package.name.here/files
```

If you try to push a local directory, or pull a remote directory, the contents of those directories will be uploaded and downloaded, respectively. However, the *directory* itself is not, which can cause some confusion.

Suppose we have a directory on our development PC named foo/. It contains four PNG files, named 1.png, 2.png, 3.png, and parallelism-is-boring.png. We then execute the following command on the command line:

```
adb push foo /storage/emulated/0/Android/data/your.package.name.here/files
```

You will wind up with:

- /storage/emulated/0/Android/data/your.package.name.here/files/1.png
- /storage/emulated/0/Android/data/your.package.name.here/files/2.png
- /storage/emulated/0/Android/data/your.package.name.here/files/3.png

- `/storage/emulated/0/Android/data/your.package.name.here/files/parallelism-is-boring.png`

Note, though, that the `foo` directory name is not included. In other words, the *contents* of `foo/` are transferred, but not `foo/` itself.

Run-As for Internal Storage

`adb push` and `adb pull` work directly for internal storage as well... on emulators.

On production hardware, though, you have some additional work to do. Specifically, you need to use external storage as an intermediary and use `adb run-as` to give yourself the temporary ability to work with internal storage.

For example, on an emulator, you could push `index.html` to the directory returned by `getFilesDir()`, for an app with an application ID of `your.package.name.here`, for the primary device account, via:

```
adb push index.html /data/data/your.package.name.here/files
```

If you try that on production hardware, it will fail. While the piece that `adb` communicates with on the emulator runs with superuser privileges, the equivalent piece on production hardware does not. The same security that prevents other apps from accessing your app's portion of internal storage prevents `adb` from doing so as well.

However, `adb` on production hardware *can* use the `run-as` command, to execute a Linux command as if it were being run by the Linux user associated with your app, the user that owns all your files and who has read/write access to those files.

So, the equivalent script to copy the file to internal storage on a production Android 4.x/5.x device would be:

```
adb push index.html /mnt/shell/emulated/0
adb shell run-as your.package.name.here cp /mnt/shell/emulated/0/index.html /data/
data/your.package.name.here/files
adb shell rm /mnt/shell/emulated/0/index.html
```

(note that the second command should appear all on one line, even though it may show up as word-wrapped here due to the length of the line and the available width of the book)

This will only work for debuggable apps, which is the normal state of apps that you run from your IDE. This script:

- Pushes the file to the root of external storage
- Uses `run-as` to run the Linux `cp` command to copy the file from external storage to the app's internal storage
- Runs the Linux `rm` command to remove the file that we placed on external storage

(if you are wondering why we do not use `mv` instead of `cp` and `rm`, `mv` generates errors related to attempting to change the ownership of the moved file)

XML Parsing Options

Android supports a fairly standard implementation of the Java DOM and SAX APIs. If you have existing experience with these, or if you have code that already leverages them, feel free to use them.

Android also bakes in the `XmlPullParser` from the xmlpull.org site. Like SAX, the `XmlPullParser` is an event-driven interface, compared to the DOM that builds up a complete data structure and hands you that result. Unlike SAX, which relies on a listener and callback methods, the `XmlPullParser` has you pull events off a queue, ignoring those you do not need and dispatching the rest as you see fit to the rest of your code.

The primary reason the `XmlPullParser` was put into Android was for XML-encoded resources. While you write plain-text XML during development, what is packaged in your APK file is a so-called "binary XML" format, where angle brackets and quotation marks and such are replaced by bitfields. This helps compression a bit, but mostly this conversion is done to speed up parsing. Android's XML resource parser can parse this "binary XML" approximately ten times faster than it can parse the equivalent plain-text XML. Hence, anything you put in an XML resource (`res/xml/`) will be parsed similarly quickly.

For plain-text XML content, the `XmlPullParser` is roughly equivalent, speed-wise, to SAX. All else being equal, lean towards SAX, simply because more developers will be familiar with it from classic Java development. However, if you really like the `XmlPullParser` interface, feel free to use it.

You are welcome to try a third-party XML parser JAR, but bear in mind that there may be issues when trying to get it working in Android.

JSON Parsing Options

Android has bundled the `org.json` classes into the SDK since the beginning, for use in parsing JSON. These classes have a DOM-style interface: you hand `JSONObject` a hunk of JSON, and it gives you an in-memory representation of the completely parsed result. This is handy but, like the DOM, a bit of a performance hog.

API Level 11 added `JSONReader`, based on Google's GSON parser, as a "streaming" parser alternative. `JSONReader` is much more reminiscent of the `XmlPullParser`, in that you pull events out of the "reader" and process them. This can have significant performance advantages, particularly in terms of memory consumption, if you do not need the entire JSON data structure. However, this is only available on API Level 11 and higher.

Because `JSONReader` is a bit "late to the party", there has been extensive work on getting other JSON parsers working on Android. Google's GSON is popular, as is Jackson. Jackson offers a few APIs, and the streaming API reportedly works very nicely on Android with top-notch performance.

Using Files with Implicit Intents

Earlier, we saw how to use an implicit `Intent` to, say, view a Web page, given an `https` URL. You can do the same sort of thing with files... though there are issues.

Technically, you can take any `File`, pass it to `Uri.fromFile()`, and get a `Uri` pointing to that file. You can put that `Uri` into an implicit `Intent`, such as one for `ACTION_VIEW`, and pass that `Intent` to `startActivity()`:

```
startActivity(new Intent(Intent.ACTION_VIEW, Uri.fromFile(somethingCool)));
```

However, at best, this only works for files on external storage. Other apps — such as whatever activity handles your `ACTION_VIEW` request — do not have rights to your portion of internal storage or your portion of removable storage. Plus, you have no guarantee that the other app has either the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permission (though, if it responded to your `Intent`, it should).

Hence, in Android 7.0, the `file` scheme on a `Uri` is banned, in effect. If you attempt to pass a `file:` `Uri` in an `Intent` that is going to another app, you will crash with a `FileUriExposedException` exception.

(you will face similar issues with putting `file:` `Uri` values on the clipboard in `ClipData` — coverage of the clipboard is later in this book)

This is coming from an updated edition of `StrictMode`. `StrictMode.VmPolicy.Builder` has a `penaltyDeathOnFileUriExposure()` method that triggers the detection of `file:` `Uri` values and the resulting `FileUriExposedException` exceptions. And, it appears that this is pre-configured, much as how `StrictMode` is pre-configured to apply `penaltyDeathOnNetwork()` (the source of your `NetworkOnMainThreadException` crashes).

However, this only kicks in if your `targetSdkVersion` is set to 24 or higher. At that point, you will need to find other ways of getting your content to other apps, such as via a class called `FileProvider`, which is covered later in this book. Or, you can also disable the check by configuring your own `StrictMode.VmPolicy` and skipping `directFileUriExposure()`, though this is not a great solution.

Visit the Trails!

In addition to this chapter, you can learn more about accessing multimedia files via the `MediaStore` and learn more about the impacts of multiple user accounts on tablets.

Tutorial #11 - Adding Simple Content

Now that we have seen how to work with assets, we can start putting them to use, by defining some "help" and "about" HTML files and displaying them in their respective activities.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Adding Some Content

In Android Studio, right-click over the `main` source set directory, choose New > Directory from the context menu, fill in the name `assets` in the dialog, and click OK. This should give you an `app/` module that looks like:

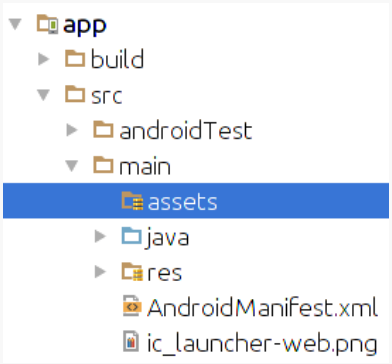


Figure 274: EmPubLite Project, Showing assets/ in main/ of app/

In `assets/`, create a `misc/` sub-folder, by right-clicking over the `assets/` folder and choosing to add a new directory named `misc` (e.g., New > Directory from the Android Studio context menu), giving you something like:

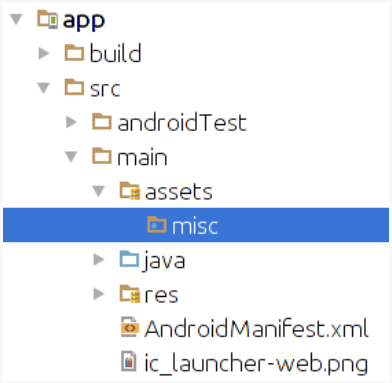


Figure 275: EmPubLite Project, Showing assets/misc/ in main/ of app/

In `assets/misc/`, create two files, `about.html` and `help.html`. In Android Studio, right-click over the `assets/misc/` folder in the project explorer, choose New > File from the context menu, fill in the desired filename in the dialog, and click OK.

The actual HTML content of these two files does not matter, so long as you can tell them apart when looking at them. If you prefer, you can download sample `about.html` and `help.html` files from the application's GitHub repository, via the links.

Step #2: Using SimpleContentFragment

Now, open up `SimpleContentActivity` and replace the stub implementation that we have now with the following Java:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;

public class SimpleContentActivity extends Activity {
  public static final String EXTRA_FILE = "file";

  @Override
  public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      String file=getIntent().getStringExtra(EXTRA_FILE);
      Fragment f=SimpleContentFragment.newInstance(file);

      getFragmentManager().beginTransaction()
        .add(android.R.id.content, f).commit();
    }
  }
}
```

(from EmPubLite-AndroidStudio/T11-HelpAbout/EmPubLite/app/src/main/java/com/commonsware/empublite/SimpleContentActivity.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

In `onCreate()`, we follow the standard recipe for defining our fragment if (and only if) we were started new, rather than restarted after a configuration change, by seeing if the fragment already exists. If we do need to add the fragment, we retrieve a string extra from the `Intent` used to launch us (identified as `EXTRA_FILE`), create an instance of `SimpleContentFragment` using that value from the extra, and execute a `FragmentTransaction` to add the `SimpleContentFragment` to our UI.

Step #3: Launching Our Activities, For Real This Time

Now, what remains is to actually supply that `EXTRA_FILE` value, which we are not doing presently when we start up `SimpleContentActivity` from `EmPubLiteActivity`.

Modify `onOptionsItemSelected()` of `EmPubLiteActivity` to look like this:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.about:
      Intent i = new Intent(this, SimpleContentActivity.class)
        .putExtra(SimpleContentActivity.EXTRA_FILE,
          "file:///android_asset/misc/about.html");
      startActivity(i);

      return(true);
```

```
      case R.id.help:
        i = new Intent(this, SimpleContentActivity.class)
          .putExtra(SimpleContentActivity.EXTRA_FILE,
            "file:///android_asset/misc/help.html");
        startActivity(i);

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from EmPubLite-AndroidStudio/T11-HelpAbout/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

You are adding the two `putExtra()` calls in the `R.id.about` and `R.id.help` branches of the `switch` statement. In both cases, we are using a quasi-URL with the prefix `file:///android_asset/`. This points to the root of our project's `assets/` folder. `WebView` knows how to interpret these URLs, to load files out of our assets directly.

Step #4: Getting a Bit More Material

Right now, our action bar on Android 5.0 devices is the one defined by `Theme.Holo.Light.DarkActionBar`. This certainly works. However, it looks a bit out of place, as most of the built-in apps will be using a material theme. So, let's make some minor adjustments to make our app blend in a bit better.

First, we need to add a `res/values-v21/` directory, representing resources that will be used solely on API Level 21+ devices. In Android Studio, right-click over the `res/` directory in your `main/` source set and choose New > "Android resource directory" from the context menu. Choose "values" as the "Resource type". Then, in the list of available qualifiers on the left, click on "Version", then click the ">>" button to the right of that list. This may give you a fairly messed-up dialog, at least in the current version of Android Studio:

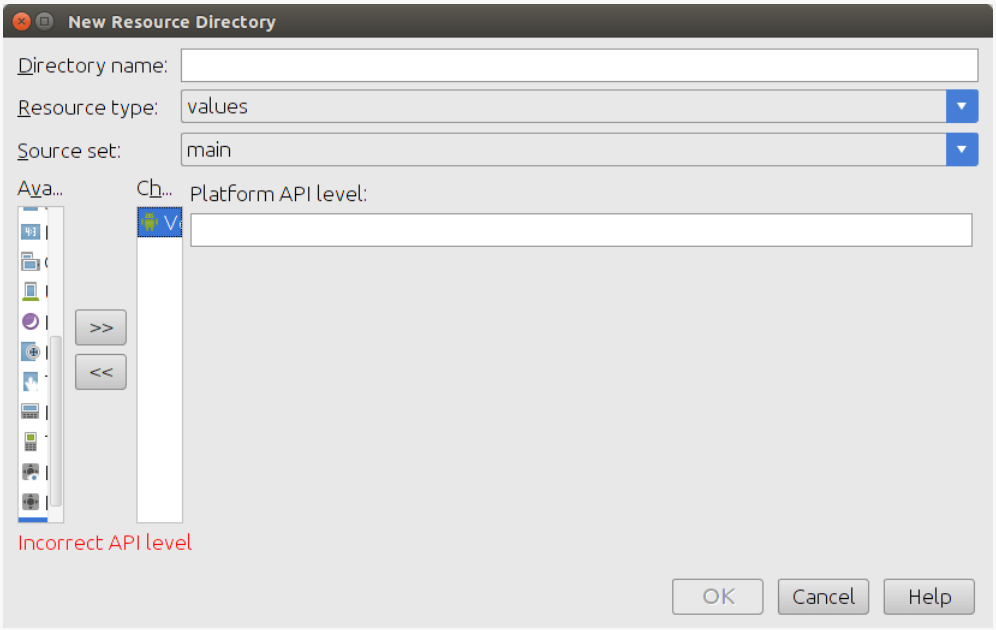


Figure 276: Android Studio New Resource Directory Dialog

Fill in `21` in the "Platform API level" field, then click OK. This should give you an empty `res/values-v21/` directory, as desired.

Then, copy the `styles.xml` file from `res/values/` into `res/values-v21/`. Windows/Linux users can drag `styles.xml` from `res/values/` while holding down the Control key to make a copy. macOS users probably have a similar convention.

Open `res/values-v21/styles.xml` and change the `parent` attribute of our one `style` element to be `android:Theme.Material.Light.DarkActionBar`. Also, add three child elements to the `<style>` element:

```
<resources>

  <!-- Base application theme. -->
  <style name="AppTheme" parent="android:Theme.Material.Light.DarkActionBar">
    <item name="android:colorPrimary">@color/colorPrimary</item>
    <item name="android:colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="android:colorAccent">@color/colorAccent</item>
  </style>

</resources>
```

(from EmPubLite-AndroidStudio/T11-HelpAbout/EmPubLite/app/src/main/res/values-v21/styles.xml)

These tell `Theme.Material` (and its descendants) to apply our existing color scheme to the theme.

Step #5: Seeing the Results

Now, if you run the application and choose "Help" from the action bar overflow, you will see your help content on-screen:

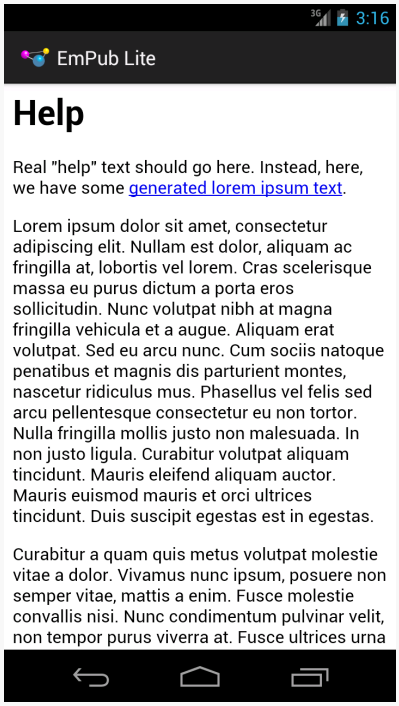


Figure 277: EmPubLite Help Screen

Pressing BACK and choosing "About" from the action bar overflow will bring up your about content:

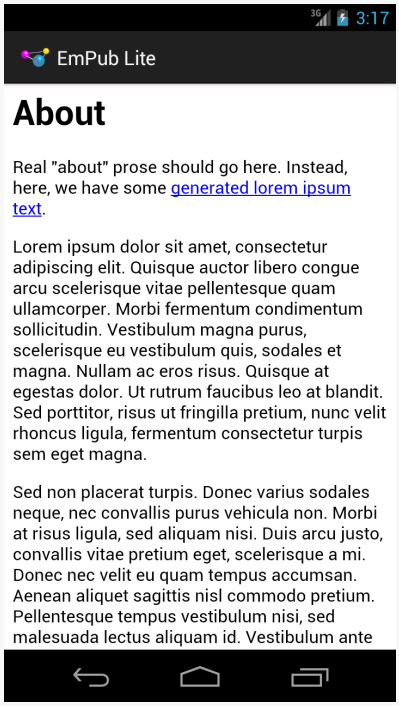


Figure 278: EmPubLite About Screen

However, on an Android 5.0 or higher device or emulator, our action bar will now sport our designated color scheme:

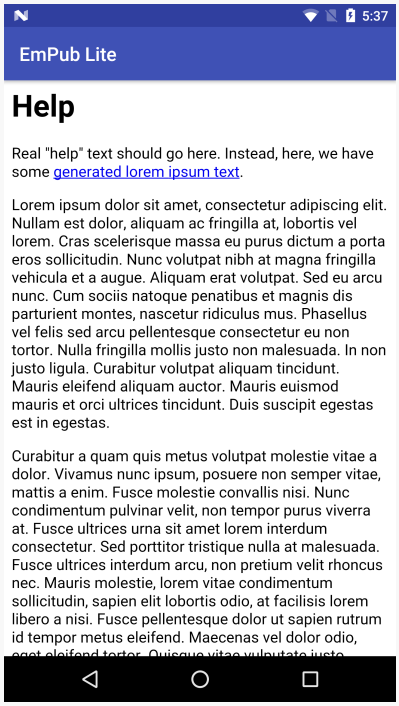


Figure 279: EmPubLite Help Screen on Android 5.1

In Our Next Episode…

… we will display the actual content of our book in our tutorial project.

Tutorial #12 - Displaying the Book

At this point, you are probably wondering when we are *ever* going to have our digital book reader let us read a digital book.

Now, in this tutorial, your patience will be rewarded.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Note that starting in this tutorial, it is assumed that you know how to add `import` statements as needed as we refer to new classes in existing code, and so the required imports are not always going to be specified.

Step #1: Adding a Book

First, we need a book. Expecting you to write a book as part of this tutorial would seem to be a bit excessive. So, instead, we will use an already-written book: The War of the Worlds, by H. G. Wells, as distributed by Project Gutenberg.

EDITOR'S NOTE: We realize that this choice of book may be seen as offensive by Martians, as it depicts them as warlike invaders with limited immune systems. Please understand that this book is a classic of Western literature and reflects the attitude of the times. If you have any concerns about this material, please contact us at *martians-so-do-not-exist@commonsware.com*.

Download http://misc.commonsware.com/WarOfTheWorlds.zip and unpack its contents (a `book/` directory of files) into your `assets/` folder of your project.

Windows and Linux Android Studio users can drag this `book/` directory into the project and drop it in `assets/` to copy the files to the proper location. You should wind up with `assets/book/` and files inside of there:

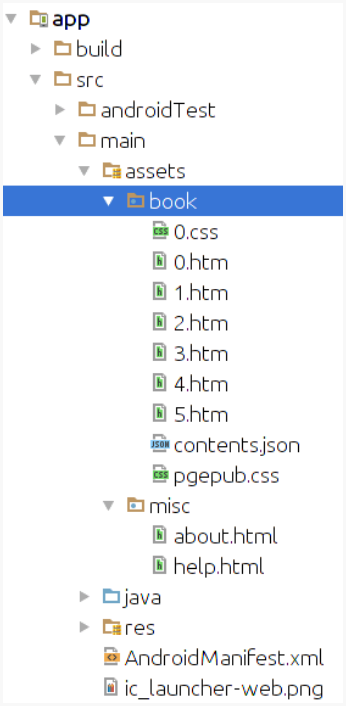


Figure 280: Android Studio Project Explorer, Showing assets/book/

In that directory, you will find some HTML and CSS files with the prose of the book, plus a `contents.json` file with metadata. We will examine this metadata in greater detail in the next section.

Step #2: Creating a ModelFragment

This sample project will use the "model fragment" pattern to hold onto the data about the book to be viewed. The "model fragment" pattern works well for cases where:

- the data is only needed by one activity, not several components, and
- we want to hold onto the data during a configuration change (e.g., screen rotation), so that we do not have to perform some work again to obtain the data

Something has to load that `BookContents`, ideally in the background, since reading an asset and parsing the JSON will take time. Also, something has to hold onto that `BookContents`, so it can be used from `EmPubLiteActivity` and the various chapter fragments in the `ViewPager`.

To that end, we will create a new class, cunningly named `ModelFragment`.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `ModelFragment` as the name, `android.app.Fragment` as the superclass, and click OK to create the empty class.

Step #3: Defining Our Model

That `contents.json` file contains a bit of metadata about the contents of the book: the book's title and a roster of its "chapters":

```
{
  "title": "The War of the Worlds",
  "chapters": [
    {
      "file": "0.htm",
      "title": "Book One: Chapters 1-9"
    },
    {
      "file": "1.htm",
      "title": "Book One: Chapters 10-14"
    },
    {
      "file": "2.htm",
      "title": "Book One: Chapters 14-17"
    },
    {
      "file": "3.htm",
      "title": "Book Two: Chapters 1-7"
    },
    {
      "file": "4.htm",
      "title": "Book Two: Chapters 7-10"
    },
    {
      "file": "5.htm",
      "title": "Project Gutenberg"
    }
```

```
  ]
}
```

In the case of this book from Project Gutenberg, the `assets/book/` directory contains six HTML files which `EmPubLite` will consider as "chapters", even though each of those HTML files contains multiple chapters from the source material. You are welcome to reorganize that HTML if you wish, updating `contents.json` to match.

We need to load `contents.json` into memory, so `EmPubLite` knows how many chapters to display and where those chapters can be found. We will pour `contents.json` into a `BookContents` model object, leveraging the GSON library that we added to our project in an earlier tutorial.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookContents` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

import java.util.List;

public class BookContents {
  List<BookContents.Chapter> chapters;

  int getChapterCount() {
    return(chapters.size());
  }

  String getChapterFile(int position) {
    return(chapters.get(position).file);
  }

  String getChapterTitle(int position) {
    return(chapters.get(position).title);
  }

  static class Chapter {
    String file;
    String title;
  }
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/BookContents.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #4: Examining Our Model

`BookContents` is a GSON interpretation of the JSON structure of `contents.json`. `BookContents` holds onto the `chapters`, as a `List` of `BookContents.Chapter` objects, each of which holds onto its file.

`BookContents` also supplies three accessor methods:

- `getChapterCount()`, to identify the number of chapters (i.e., the size of the `chapters` array in the JSON)
- `getChapterFile()`, to return the relative path within `assets/book/` that represents our "chapter" of HTML
- `getChapterTitle()`, to return the title of this "chapter" of the book

Step #5: Defining Our Event

We will want to load the JSON and create the `BookContents` on a background thread, as we will be performing enough I/O and parsing that we might make our UI a bit sluggish if we do the work on the main application thread. However, we need to let the UI layer (`EmPubLiteActivity` and its `ViewPager`) know when the book is loaded, so it can be poured into the user interface.

We could use an `AsyncTask` for that, notifying the activity in `onPostExecute()`. However, we will need more flexible inter-component communication over time, things that cannot be handled by a simple `AsyncTask`. Hence, we will start using the event bus pattern here, employing greenrobot's EventBus library that we added to our project in a previous tutorial.

With EventBus, we create our own event classes. The one event that we have up front is one to indicate that our book metadata has been loaded and is ready for use, in the form of a `BookContents` object. Hence, in this step of the tutorial, we will define a `BookLoadedEvent` that will be posted when the book is loaded. And, we will have the event hold onto the `BookContents`, to lightly simplify populating the UI later on.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookLoadedEvent` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

public class BookLoadedEvent {
  private BookContents contents=null;

  public BookLoadedEvent(BookContents contents) {
    this.contents=contents;
  }

  public BookContents getBook() {
    return(contents);
  }
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/BookLoadedEvent.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #6: Loading Our Model

Now, we need to actually arrange to load the book on a background thread and post our newly-created `BookLoadedEvent`. This is one of the key jobs of our `ModelFragment`: to manage the loading of our activity's model, using background threads.

With that in mind, replace our stub `ModelFragment` implementation with the following:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.os.Process;
import android.util.Log;
import com.google.gson.Gson;
```

```
import org.greenrobot.eventbus.EventBus;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.concurrent.atomic.AtomicReference;

public class ModelFragment extends Fragment {
  final private AtomicReference<BookContents> contents=
    new AtomicReference<>();

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
  }

  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    if (contents.get()==null) {
      new LoadThread(host.getAssets()).start();
    }
  }

  public BookContents getBook() {
    return(contents.get());
  }

  private class LoadThread extends Thread {
    private AssetManager assets=null;

    LoadThread(AssetManager assets) {
      super();

      this.assets=assets;
    }

    @Override
    public void run() {
      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
      Gson gson=new Gson();

      try {
        InputStream is=assets.open("book/contents.json");
        BufferedReader reader=
          new BufferedReader(new InputStreamReader(is));
```

```
        contents.set(gson.fromJson(reader, BookContents.class));

        EventBus.getDefault().post(new BookLoadedEvent(getBook()));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
    }
  }
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

In `onCreate()`, we call `setRetainInstance(true)`, to tell the framework to keep this fragment despite a configuration change, just passing it to the new activity created as a result of that configuration change.

In `onAttach()`, if we do not already have our `BookContents` object, we fork a `LoadThread` to populate it, and we cannot readily get at an `AssetManager` until we are attached to the hosting activity. This is why we are not forking `LoadThread` in `onCreate()`. You may see this method name appear with strikethrough formatting. `onAttach()` taking a `Context` as a parameter was added in API Level 23, and `onAttach()` taking a `Activity` as a parameter was deprecated. However, our `minSdkVersion` is lower than 23, so we need to use the older callback method.

`LoadThread` takes the `AssetManager` as a parameter, stashing it in a field in the `LoadThread` constructor.

Then, in the `run()` method that is called on the background thread, we call `setThreadPriority()` to drop the thread's priority to that of a background thread. This reduces how much we compete with the main application thread for CPU time. Then, we read in the JSON using GSON to create the `BookContents` instance. GSON automatically de-serializes our JSON into the `BookContents` and `BookContents.Chapter` instances, given that we are telling the `fromJson()` method that it is to be loading an instance of a `BookContents` object. Finally, we `post()` a `BookLoadedEvent` to the default `EventBus`.

The `BookContents` is wrapped in an `AtomicReference`, in case the main application thread tries to get the `BookContents` at the same time our background thread tries to

set that field's value. Using an `AtomicReference` handles our thread synchronization for us.

The `open()` method on `AssetManager` could throw an `IOException`. Normally, this indicates a development-time bug (e.g., we failed to actually set up the `book/contents.json` file), which is why we log the message to Logcat. A production-grade book reader should also `post()` an `EventBus` event to allow the UI layer to let the user know that we could not load the book. As it stands, the book reader will remain stuck on the `ProgressBar` forever in case of this sort of problem. Augmenting the tutorial in this way is left as an exercise for the reader.

Note that the `LoadThread` implementation has a pair of references to `Process`. In this case, this is `android.os.Process`, not `java.lang.Process`. Since `java.lang.Process` is automatically imported, if you fail to import `android.os.Process`, you will see errors about how `THREAD_PRIORITY_BACKGROUND` and `setThreadPriority()` are not defined. Since we are not using `java.lang.Process` in this class, having the import to `android.os.Process` (as shown in the code listing above) resolves this conflict.

Step #7: Registering for Events

Right now, our `BookLoadedEvent` will be posted... and ignored, as nothing in the application is set up to watch for such events. Our `EmPubLiteActivity` needs to know about these events, and the first step to accomplishing that is to have it register for events in general with the `EventBus`.

Add the following two methods to `EmPubLiteActivity`:

```
@Override
  public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);
  }

  @Override
  public void onStop() {
    EventBus.getDefault().unregister(this);
    super.onStop();
  }
```

These simply register the activity with the `EventBus` while it is in the foreground.

Step #8: Adapting the Content

Before we can use the `BookContents`, we need to update `ContentsAdapter` to display the prose on the screen.

First, add a `BookContents` data member to `ContentsAdapter`:

```
  final BookContents contents;
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

Then, add the `BookContents` parameter to the constructor, assigning it to the new data member:

```
  public ContentsAdapter(Activity ctxt, BookContents contents) {
    super(ctxt.getFragmentManager());

    this.contents=contents;
  }
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

Next, update `getCount()` to use the `getChapterCount()` of our `BookContents`:

```
  @Override
  public int getCount() {
    return(contents.getChapterCount());
  }
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

Then, modify `getItem()` to retrieve the relative path for a given chapter from the `BookContents` and create a `SimpleContentFragment` on the complete `file:///android_asset` path to the file in question:

```
  @Override
  public Fragment getItem(int position) {
    String path=contents.getChapterFile(position);

    return(SimpleContentFragment.newInstance("file:///android_asset/book/"
      + path));
  }
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

Note that you may need to change the parameter name in the `getItem()` declaration to be `position`, as it may be another value (e.g., `arg0`).

Finally, add `getPageTitle()`, pulling our tab title from the chapter title:

```
@Override
public CharSequence getPageTitle(int position) {
  return(contents.getChapterTitle(position));
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

Step #9: Showing the Content When Loaded

Now, we can actually add the logic to display the book once it is loaded.

Create a `setupPager()` method on `EmPubLiteActivity` as follows:

```
private void setupPager(BookContents contents) {
  adapter=new ContentsAdapter(this, contents);
  pager.setAdapter(adapter);

  MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
  tabs.setViewPager(pager);
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

The contents of this method are almost identical to some lines in `onCreate()` – we have just moved them to a separate method. Remove those duplicate lines from `onCreate()`, so you have:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  pager=(ViewPager)findViewById(R.id.pager);
}
```

Then, add the following `onBookLoaded()` method to `EmPubLiteActivity`:

```
@SuppressWarnings("unused")
@Subscribe(threadMode =ThreadMode.MAIN)
public void onBookLoaded(BookLoadedEvent event) {
```

```
    setupPager(event.getBook());
  }
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

This tells `EventBus` that if a `BookLoadedEvent` is posted, we are interested in it, and it should be delivered to our `onBookLoaded()` method on the main application thread. This method looks like it is unused, because it will be called using reflection by the EventBus, and the IDE does not know that. The `@SuppressWarnings("unused")` annotation indicates that this method is used.

Step #10: Attaching our ModelFragment

We also need to add some code to set up the `ModelFragment` — it will not magically appear on its own. So, the first time we create an `EmPubLiteActivity`, we want to create our `ModelFragment`. To do that, define a static data member named `MODEL` in `EmPubLiteActivity`:

```
  private static final String MODEL="model";
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Then, update the `onStart()` method in `EmPubLiteActivity` to see if we already have the fragment before creating one:

```
@Override
public void onStart() {
  super.onStart();
  EventBus.getDefault().register(this);

  if (adapter==null) {
    ModelFragment mfrag=
        (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (mfrag == null) {
      getFragmentManager().beginTransaction()
          .add(new ModelFragment(), MODEL).commit();
    }
  }
}
```

If you run the result in a device or emulator, you will see the book content appear:

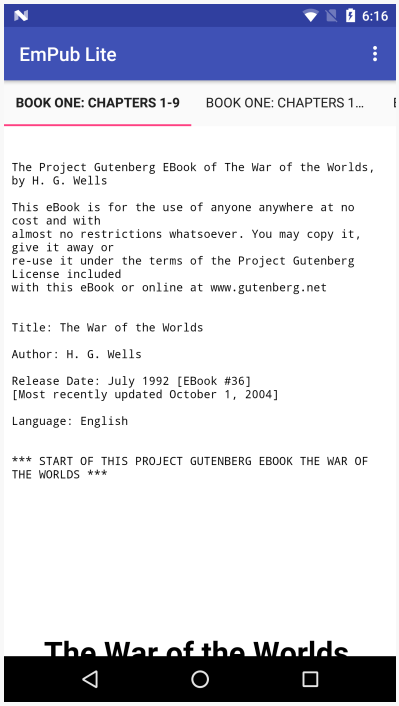


Figure 281: EmPubLite, With Content

Swiping left and right will take you to the other portions of the book.

Step #11: Showing the Content After a Configuration Change

While you can see the book contents now, if you try rotating the screen, the book contents will not appear. That is because the `ModelFragment` has already loaded the contents (so the `BookLoadedEvent` has passed), but we have no logic in `EmPubLiteActivity` to populate the book by other means.

To do that, simply add an `else if` clause to the `if` in `onStart()`, to get the book contents over to `setupPager()` if they are ready:

```
@Override
public void onStart() {
  super.onStart();
  EventBus.getDefault().register(this);

  if (adapter==null) {
    ModelFragment mfrag=
```

```
      (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (mfrag==null) {
      getFragmentManager().beginTransaction()
        .add(new ModelFragment(), MODEL).commit();
    }
    else if (mfrag.getBook()!=null) {
      setupPager(mfrag.getBook());
    }
  }
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Now, if you run the sample and rotate the screen (e.g., Ctrl-Right on the Windows/Linux emulator), the book will appear in either case.

Step #12: Setting Up StrictMode

Since we are now starting to do disk I/O, particularly aiming to have it done on background threads, it would be a good idea to configure `StrictMode`, so it will complain if we fail in our quest and accidentally do this I/O on the main application thread.

Add the following method to `EmPubLiteActivity`:

```
private void setupStrictMode() {
  StrictMode.ThreadPolicy.Builder builder=
    new StrictMode.ThreadPolicy.Builder()
      .detectAll()
      .penaltyLog();

  if (BuildConfig.DEBUG) {
    builder.penaltyFlashScreen();
  }

  StrictMode.setThreadPolicy(builder.build());
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Here, we create a `StrictMode.ThreadPolicy.Builder`, configured to detect all violations on the main application thread, logging them to Logcat. In addition, if we are in a `DEBUG` build, we will flash a red border around the screen.

Note, though, that this red border will appear even if *we* do not make any mistakes. Unfortunately, Google engineers do not check the framework code for these sorts of violations, leading to some bugs that we as app developers cannot resolve. Those will be reported as `StrictMode` violations, just as if we had made the mistakes ourselves.

At the present time, this tutorial does not trigger any `StrictMode` violations, and so the red border flash should not appear. However, changes in newer versions of Android, or newer versions of the support libraries, might change that, at which time the red flashes will point out that the author of this book has to fix the tutorials.

Then, just after `super.onCreate()` in the `onCreate()` method in `EmPubLiteActivity`, add in a call to the new `setupStrictMode()` method. This will give you an `onCreate()` method that looks like:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  setupStrictMode();
  pager=(ViewPager)findViewById(R.id.pager);
}
```

(from EmPubLite-AndroidStudio/T12-Book/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

In Our Next Episode…

… we will allow the user to manipulate some preferences in our tutorial project.

Using Preferences

Android has many different ways for you to store data for long-term use by your activity. The simplest ones to use are `SharedPreferences` and simple files.

Android allows activities and applications to keep preferences, in the form of key/value pairs (akin to a `Map`), that will hang around between invocations of an activity. As the name suggests, the primary purpose is for you to store user-specified configuration details, such as the last feed the user looked at in your feed reader, or what sort order to use by default on a list, or whatever. Of course, you can store in the preferences whatever you like, so long as it is keyed by a `String` and has a primitive value (`boolean`, `String`, etc.)

Preferences can either be for a single activity or shared among all activities in an application. Other components, such as services, also can work with shared preferences.

Getting What You Want

To get access to the preferences, you have three APIs to choose from:

- `getPreferences()` from within your `Activity`, to access activity-specific preferences
- `getSharedPreferences()` from within your `Activity` (or other application `Context`), to access application-level preferences
- `getDefaultSharedPreferences()`, on `PreferenceManager`, to get the shared preferences that work in concert with Android's overall preference framework

The first two take a security mode parameter. The right answer here is `MODE_PRIVATE`, so no other applications can access the file. The `getSharedPreferences()` method also takes a name of a set of preferences; `getPreferences()` effectively calls `getSharedPreferences()` with the activity's class name as the preference set name. The `getDefaultSharedPreferences()` method takes the `Context` for the preferences (e.g., your `Activity`).

All of those methods return an instance of `SharedPreferences`, which offers a series of getters to access named preferences, returning a suitably-typed result (e.g., `getBoolean()` to return a boolean preference). The getters also take a default value, which is returned if there is no preference set under the specified key.

Unless you have a good reason to do otherwise, you are best served using the third option above — `getDefaultSharedPreferences()` — as that will give you the `SharedPreferences` object that works with a `PreferenceActivity` by default, as will be described later in this chapter.

Stating Your Preference

Given the appropriate `SharedPreferences` object, you can use `edit()` to get an "editor" for the preferences. This object has a set of setters that mirror the getters on the parent `SharedPreferences` object. It also has:

1. `remove()` to get rid of a single named preference
2. `clear()` to get rid of all preferences
3. `apply()` or `commit()` to persist your changes made via the editor

The last one is important — if you modify preferences via the editor and fail to save the changes, those changes will evaporate once the editor goes out of scope. `commit()` is a blocking call, while `apply()` works asynchronously. Ideally, use `apply()` where possible, though it was only added in Android 2.3, so it may not be available to you if you are aiming to support earlier versions of Android than that.

Conversely, since the preferences object supports live changes, if one part of your application (say, an activity) modifies shared preferences, another part of your application (say, a service) will have access to the changed value immediately.

Collecting Preferences with PreferenceFragment

Some "preferences" will be collected as part of the natural use of your user interface. For example, if you have a `SeekBar` to control a zoom level, you might elect to record the `SeekBar` position in `SharedPreferences`, so you can restore the user's last zoom level later on.

However, in many cases, we have various settings that we would like the user to be able to configure but are not something that the user would configure elsewhere in our UI. You could roll your own UI to collect preferences in bulk from the user. On the whole, this is a bad idea. Instead, use preference XML resources and a `PreferenceFragment`.

Why?

One of the common complaints about Android developers is that they lack discipline, not following any standards or conventions inherent in the platform. For other operating systems, the device manufacturer might prevent you from distributing apps that violate their human interface guidelines. With Android, that is not the case — but this is not a blanket permission to do whatever you want. Where there is a standard or convention, please follow it unless you have a *clear* reason not to, so that users will feel more comfortable with your app and their device.

Using a `PreferenceFragment` for collecting preferences is one such convention.

The linchpin to the preferences framework and `PreferenceFragment` is yet another set of XML data structures. You can describe your application's preferences in XML files stored in your project's `res/xml/` directory. Given that, Android can present a UI for manipulating those preferences, one which matches what you see in the Settings app. The user's choices are then stored in the `SharedPreferences` that you get back from `getDefaultSharedPreferences()`.

This can be seen in the `Prefs/Fragment` sample project.

Showing the Current Values

This project's main activity hosts a `TableLayout`, into which we will load the values of five preferences:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/checkbox"/>

    <TextView
      android:id="@+id/checkbox"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/ringtone"/>

    <TextView
      android:id="@+id/ringtone"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/text"/>

    <TextView
      android:id="@+id/text"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/list"/>

    <TextView
      android:id="@+id/list"
      style="@style/value"/>
  </TableRow>
```

```
</TableLayout>
```

(from Prefs/Fragment/app/src/main/res/layout/content.xml)

The above layout is used by `PreferenceContentsFragment`, which populates the right-hand column of `TextView` widgets at runtime in `onResume()`, pulling the values from the default `SharedPreferences` for our application:

```
package com.commonsware.android.preffrag;

import android.app.Fragment;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class PreferenceContentsFragment extends Fragment {
  private TextView checkbox=null;
  private TextView ringtone=null;
  private TextView text=null;
  private TextView list=null;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.content, parent, false);

    checkbox=(TextView)result.findViewById(R.id.checkbox);
    ringtone=(TextView)result.findViewById(R.id.ringtone);
    text=(TextView)result.findViewById(R.id.text);
    list=(TextView)result.findViewById(R.id.list);

    return(result);
  }

  @Override
  public void onResume() {
    super.onResume();

    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(getActivity());

    checkbox.setText(Boolean.valueOf(prefs.getBoolean("checkbox", false)).toString());
    ringtone.setText(prefs.getString("ringtone", "<unset>"));
    text.setText(prefs.getString("text", "<unset>"));
    list.setText(prefs.getString("list", "<unset>"));
  }
}
```

(from Prefs/Fragment/app/src/main/java/com/commonsware/android/preffrag/PreferenceContentsFragment.java)

The main activity, `FragmentsDemo`, simply loads `res/layout/main.xml`, which contains a `<fragment>` element pointing at `PreferenceContentsFragment`. It also defines an options menu, which we will examine later in this section.

The result is an activity showing the default values of the preferences when it is first run, since we have not set any values yet:

Figure 282: Activity Showing Preference Values

Defining Your Preferences

First, you need to tell Android what preferences you are trying to collect from the user.

To do this, you will need to add a `res/xml/` directory to your project, if one does not already exist. Then, for your `PreferenceFragment`, you will define one of these XML resource files. The root element of this XML file will be `<PreferenceScreen>`, and it will contain child elements, one per preference.

In the sample project, we have one such file, `res/xml/preferences.xml`:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

  <CheckBoxPreference
    android:key="checkbox"
    android:summary="@string/pref1summary"
    android:title="@string/pref1title"/>

  <RingtonePreference
    android:key="ringtone"
    android:showDefault="true"
    android:showSilent="true"
    android:summary="@string/pref2summary"
    android:title="@string/pref2title"/>

  <EditTextPreference
    android:dialogTitle="@string/dialogtitle"
    android:key="text"
    android:summary="@string/pref3summary"
    android:title="@string/pref3title"/>

  <ListPreference
    android:dialogTitle="@string/listdialogtitle"
    android:entries="@array/cities"
    android:entryValues="@array/airport_codes"
    android:key="list"
    android:summary="@string/pref4summary"
    android:title="@string/pref4title"/>

</PreferenceScreen>
```

(from Prefs/Fragment/app/src/main/res/xml/preferences.xml)

Each preference element has two attributes at minimum:

1. `android:key`, which is the key you use to look up the value in the `SharedPreferences` object via methods like `getInt()`
2. `android:title`, which is a few words identifying this preference to the user

You may also wish to consider having `android:summary`, which is a short sentence explaining what the user is to supply for this preference.

There are lots of other attributes that are common to all preference elements, and there are more types of preference elements than the ones that we used in the preference XML shown above. We will examine more preference elements later in this chapter.

Creating Your PreferenceFragment

Preference XML, on API Level 11 and higher, is loaded by an implementation of `PreferenceFragment`. The mission of `PreferenceFragment` is to call `addPreferencesFromResource()` in `onCreate()`, supplying the resource ID of the preference XML to load (e.g., `R.xml.preference2`). That fragment, in turn, can be loaded up by a simple `Activity`.

In fact, the fragment is so short, you could even make it be a static class inside the activity, as is done in the sample app. The activity that collects the preferences, `EditPreferences`, has a `Prefs` static subclass of `PreferenceFragment`:

```
package com.commonsware.android.preffrag;

import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceFragment;

public class EditPreferences extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager().beginTransaction()
          .add(android.R.id.content,
              new Prefs()).commit();
    }
  }

  public static class Prefs extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      addPreferencesFromResource(R.xml.preferences);
    }
  }
}
```

(from Prefs/Fragment/app/src/main/java/com/commonsware/android/preffrag/EditPreferences.java)

The only thing that `Prefs` does is call the inherited `addPreferencesFromResource()` in its `onCreate()` method, supplying the ID of the preference XML. All

`EditPreferences` does is show the fragment, in this case using a `FragmentTransaction`.

The Results

An action bar item in `MainActivity` starts up the `EditPreferences` activity. If you click that from the overflow, you will see the UI created from your XML by means of the `PreferenceFragment`:

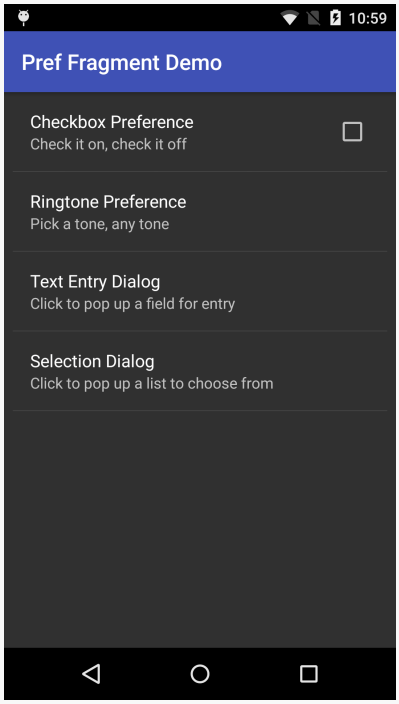


Figure 283: Activity Collecting Preference Values

If you make a change, such as tapping on the checkbox, and press BACK to return to the original activity, you will see the resulting change in the preference values themselves:

Figure 284: Original Activity, Showing Revised Preference Value

Android Studio's Preferences Editor

If you open up a preference XML resource in Android Studio, you will be given an editor that is reminiscent of the layout resource editor. You will have two sub-tabs: "Text" with the XML and "Design" with a drag-and-drop UI:

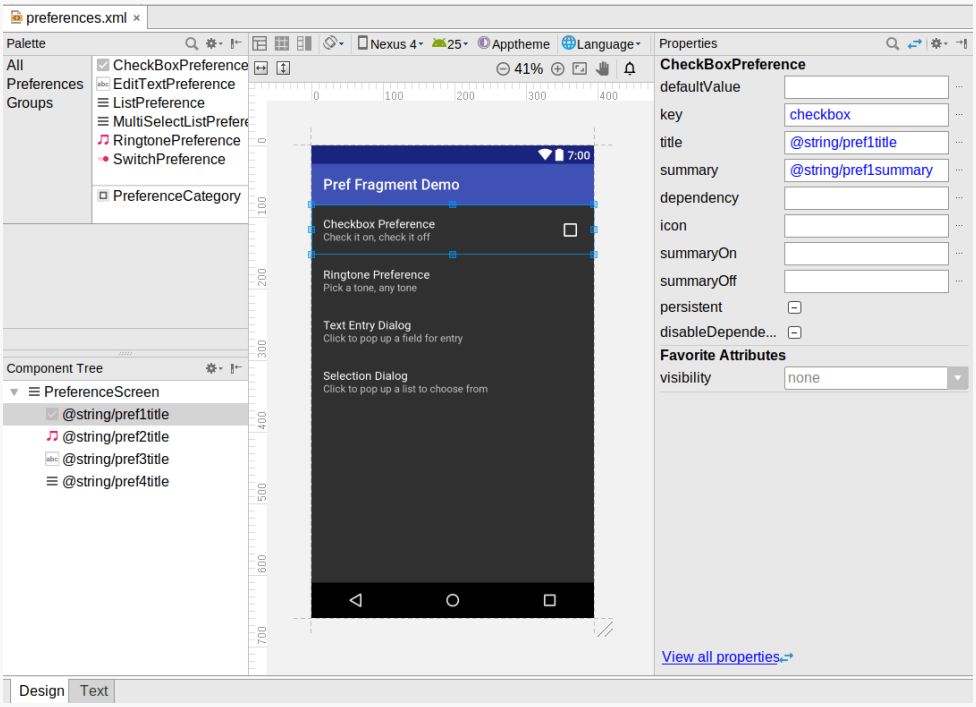


Figure 285: Android Studio Preferences Editor

The drag-and-drop editor UI works akin to its layout resource editor counterpart. You can drag a preference from the Palette into either the preview area or into the Component Tree to add it to the resource. For any selected preference, the Attributes pane allows you to modify attributes, either from the default short list of popular properties or the full list of properties that you get from clicking "View all properties".

Types of Preferences

There are a variety of subclasses of `Preference` in the Android SDK for use with `PreferenceActivity`. This section will outline the major ones. Later in the book we will examine how to create your own custom `Preference` classes.

CheckBoxPreference and SwitchPreference

The sample application shown above a `CheckBoxPreference`. A `CheckBoxPreference` is an "inline" preference, in that the widget the user interacts with (in this case, a `CheckBox`) is part of the preference screen itself, rather than contained in a separate dialog.

`SwitchPreference` is functionally equivalent to `CheckBoxPreference`, insofar as both collect `boolean` values from the user. The difference is that `SwitchPreference` uses a `Switch` widget that the user slides left and right to toggle between "on" and "off" states. Also note that `SwitchPreference` was added in API Level 14 and therefore will not be available to older Android versions.

EditTextPreference

`EditTextPreference`, when tapped by the user, pops up a dialog that contains an `EditText` widget. You can configure this widget via attributes on the `<EditTextPreference>` element — in addition to standard preference attributes like `android:key`, you can include any attribute understood by `EditText`, such as `android:inputType`.

The value stored in the `SharedPreferences` is a string.

The sample app has an `EditTextPreference`:

```
<EditTextPreference
  android:dialogTitle="@string/dialogtitle"
  android:key="text"
  android:summary="@string/pref3summary"
  android:title="@string/pref3title"/>
```

(from Prefs/Fragment/app/src/main/res/xml/preferences.xml)

When the user taps on it in the `PreferenceFragment`, the user will see a dialog where they can fill in a value, or edit an existing value if they provided one previously:

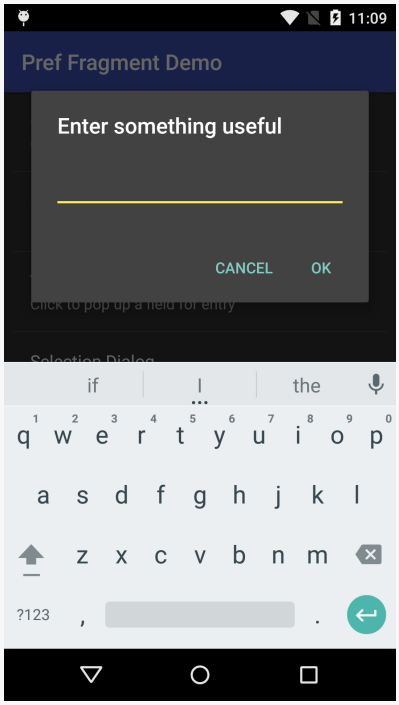


Figure 286: EditTextPreference UI

RingtonePreference

`RingtonePreference` pops up a dialog with a list of ringtones installed on the device or emulator. However, bear in mind that older emulator images may not have any pre-installed ringtones.

In addition to the standard preference attributes, you can include `android:showDefault`, indicating that the list should contain a "Default ringtone" option. If the user chooses this ringtone, they are effectively choosing the same ringtone that they have set up for incoming phone calls.

You can also use `android:showSilent`, which allows the user to choose a "Silence" pseudo-ringtone, to indicate not to play any ringtone.

The sample app has a `RingtonePreference`:

```
<RingtonePreference
  android:key="ringtone"
  android:showDefault="true"
  android:showSilent="true"
```

```
    android:summary="@string/pref2summary"
    android:title="@string/pref2title"/>
```

(from Prefs/Fragment/app/src/main/res/xml/preferences.xml)

When the user taps on it in the `PreferenceFragment`, the user will see a roster of ringtones, along with "Default" and "None" options, since we opted into those:

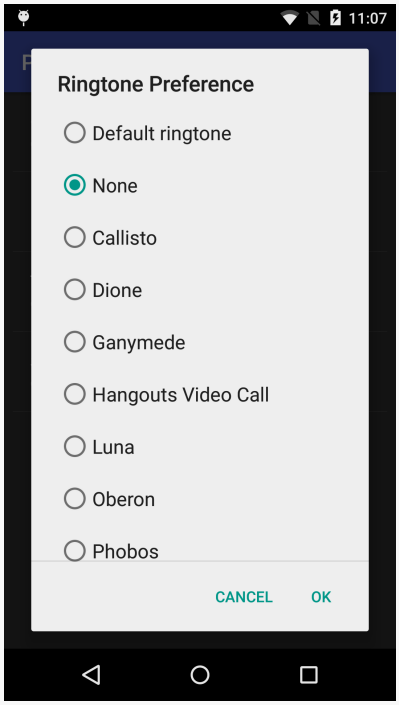


Figure 287: RingtonePreference UI

The value stored in the `SharedPreferences` is a string, specifically the string representation of a `Uri` pointing to a `ContentProvider` that can serve up the ringtone for playback. The use of `ContentProvider` will be covered in a later chapter, and playing back media like ringtones will be covered in another later chapter.

ListPreference and MultiSelectListPreference

Visually, a `ListPreference` looks just like `RingtonePreference`, except that *you* control what goes into the list. You do this by specifying a pair of `string-array` resources in your preference XML.

String resources hold individual strings; string array resources hold a collection of strings. Typically, you will find string array resources in `res/values/arrays.xml` and

related resource sets for translation. The `<string-array>` element has the `name` attribute to identify the resource, along with child `<item>` elements for the individual strings in the array.

So, our sample app has a pair of `<string-array>` resources in `res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="cities">
    <item>Philadelphia</item>
    <item>Pittsburgh</item>
    <item>Allentown/Bethlehem</item>
    <item>Erie</item>
    <item>Reading</item>
    <item>Scranton</item>
    <item>Lancaster</item>
    <item>Altoona</item>
    <item>Harrisburg</item>
  </string-array>
  <string-array name="airport_codes">
    <item>PHL</item>
    <item>PIT</item>
    <item>ABE</item>
    <item>ERI</item>
    <item>RDG</item>
    <item>AVP</item>
    <item>LNS</item>
    <item>AOO</item>
    <item>MDT</item>
  </string-array>
</resources>
```

(from Prefs/Fragment/app/src/main/res/values/arrays.xml)

Here, the actual strings are written in-line. They could just as easily be references to string resource (e.g., `<item>@string/philly</item>`). For user-facing strings, like those in the `cities` array, having them as string resources may make it easier for you to manage your translations.

The sample app then uses those arrays in a `ListPreference`:

```
  <ListPreference
    android:dialogTitle="@string/listdialogtitle"
    android:entries="@array/cities"
    android:entryValues="@array/airport_codes"
```

```
    android:key="list"
    android:summary="@string/pref4summary"
    android:title="@string/pref4title"/>
```

(from Prefs/Fragment/app/src/main/res/xml/preferences.xml)

This then allows the user to choose a city, when the user taps on this preference in the `PreferenceFragment`:

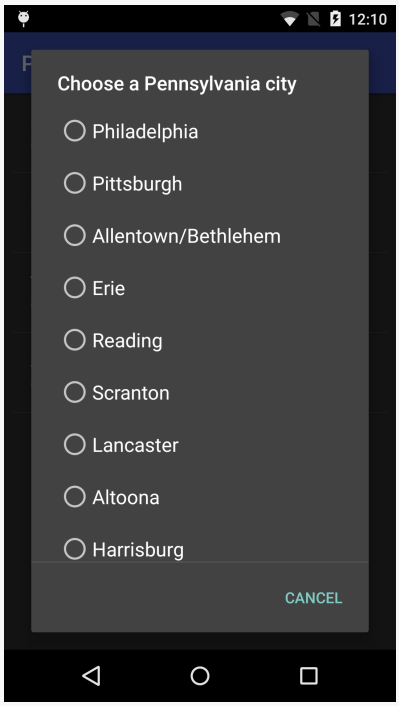


Figure 288: ListPreference UI

However, when the user chooses a city by name (e.g., Philadelphia), what is stored in the `SharedPreferences` is the *corresponding* airport code (e.g., PHL).

`MultiSelectListPreference` works much the same way, except:

- The list contains checkboxes, not radio buttons
- The user can check multiple items
- The result is stored in a "string set" in the `SharedPreferences`, retrieved via `getStringSet()`
- It is only available on API Level 11 and higher

We will see `MultiSelectListPreference` in action later in the book.

Tutorial #13 - Using Some Preferences

Now that we have the core reading functionality working, we can start to add other features for the user.

One common thing in Android applications is to collect preferences from the user, tailoring the way the app behaves. In the case of `EmPubLite`, we will initially track two preferences:

- Whether the user wants to return to the book on the same chapter (page in the `ViewPager`) that they were on when they last were reading the book
- Whether the user wants us to keep the screen on, so they do not have to keep tapping the screen to prevent Android's automatic sleep mode from kicking in

In this tutorial, we will collect and use these two preferences.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Defining the Preference XML Files

We need an XML resource file to define what preferences we wish to collect.

First, add four new `<string>` elements to `res/values/strings.xml`:

```
<string name="lastposition_title">Save Last Position</string>
<string name="lastposition_summary">Save the last chapter you were viewing and open
up on that chapter when re-opening the app</string>
```

```
<string name="keepscreenon_summary">Keep the screen powered on while the reader is in
the foreground</string>
<string name="keepscreenon_title">Keep Screen On</string>
```

Next, right click over `res/` in your project, and choose New > "Android resource directory" from the context menu. Change the "Resource type" drop-down to be "xml", then click OK to create the directory.

Then, right-click over your new `res/xml/` directory and choose New > "XML resource file" from the context menu. Fill in `pref_display.xml` in the "New XML Resource File" dialog, then click OK to create the file.

This will open up in a preference screen editor, with "Design" and "Text" sub-tabs:

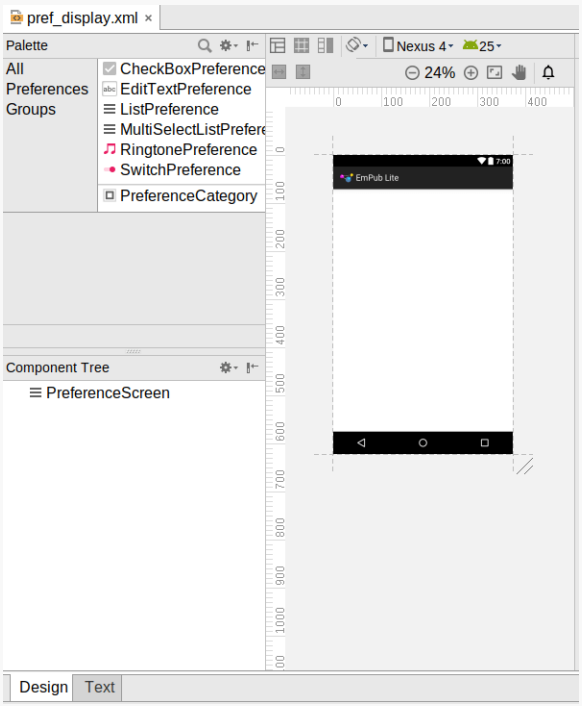


Figure 289: Android Studio Preference Screen Editor, As Initially Launched

Drag a `CheckBoxPreference` from the Palette into the preview area. Then, in the Attributes pane, set the key to `saveLastPosition`, set the title to `@string/lastposition_title`, and set the summary to `@string/lastposition_summary`:

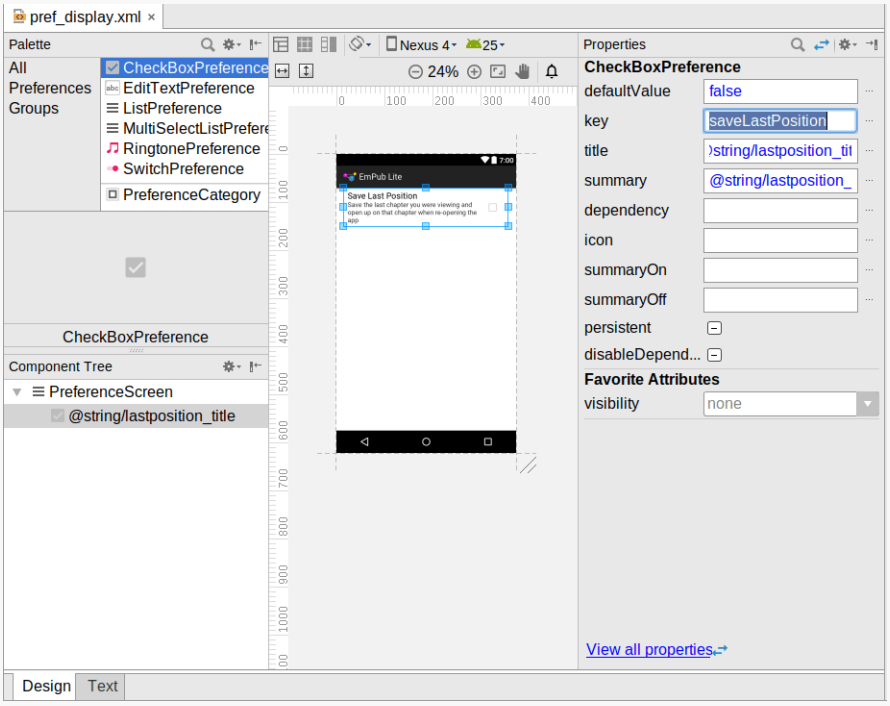


Figure 290: Android Studio Preference Screen Editor, After First Preference

Next, drag another `CheckBoxPreference` from the Palette into the preview area. Then, in the Attributes pane for this newly-added preference, set the key to `keepScreenOn`, set the title to `@string/keepscreenon_title`, and set the summary to `@string/keepscreenon_summary`:

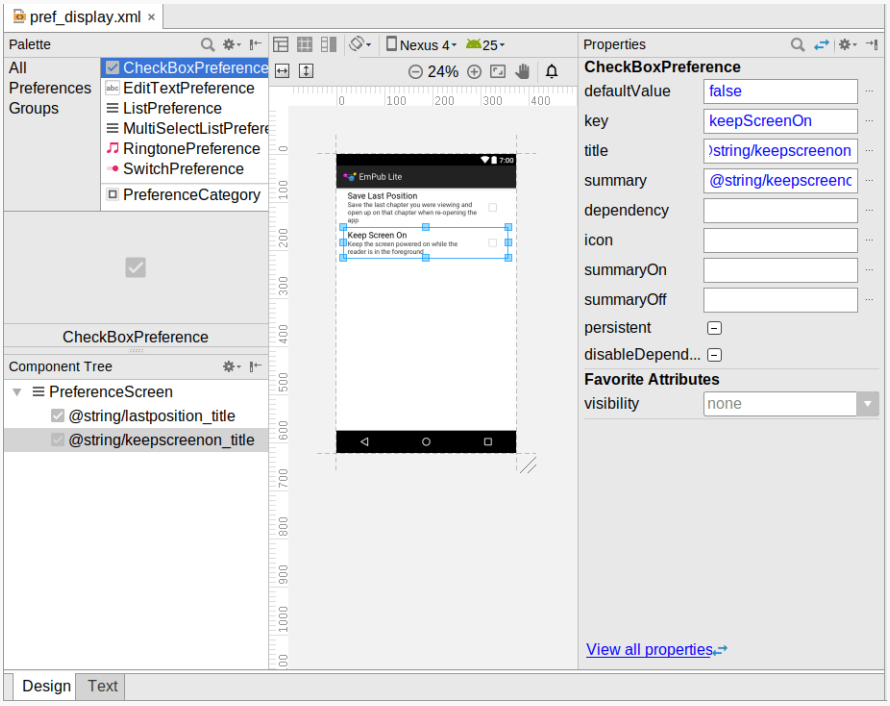


Figure 291: Android Studio Preference Screen Editor, After Second Preference

If you look at the XML in the "Text" sub-tab, you should see that it resembles:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">

  <CheckBoxPreference
    android:defaultValue="false"
    android:title="@string/lastposition_title"
    android:key="saveLastPosition"
    android:summary="@string/lastposition_summary" />
  <CheckBoxPreference
    android:defaultValue="false"
    android:title="@string/keepscreenon_title"
    android:key="keepScreenOn"
    android:summary="@string/keepscreenon_summary" />
</PreferenceScreen>
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/res/xml/pref_display.xml)

Step #2: Creating Our Preference Activity

We will eventually load that preference XML into a `PreferenceFragment`. We could use a `PreferenceActivity` for that, but we do not have enough preferences to warrant a full master/detail setup. Instead, we can just display the `PreferenceFragment` in a regular `Activity`, named `Preferences`, using a static inner class implementation of a `PreferenceFragment`, named `Display`.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Activity > Empty Activity from the context menu. Set the activity name to be `Preferences` and uncheck all the checkboxes. Then click Finish to add the activity to the project.

In the `Preferences` class that is created, replace the current implementation with the following:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;
import android.preference.PreferenceFragment;

public class Preferences extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager()
        .beginTransaction()
        .add(android.R.id.content, new Display())
        .commit();
    }
  }

  public static class Display extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.pref_display);
    }
  }
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/Preferences.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #3: Adding To Our Action Bar

Of course, having this activity does us no good if we cannot start it up, so we need to add another hook to our action bar configuration for that.

First, add a `settings` string resource, with a value of `Settings`:

```
<string name="settings">Settings</string>
```

Then, right-click over the `res/` directory of your `app/` module, and choose New > Vector Asset from the context menu. Click the Icon button, search for `settings`, and choose the "settings" icon, creating an `ic_settings_black_24dp` icon.

Click Next, then Finish.

Finally, add the following XML element to `res/menu/options.xml` as the first child of the `<menu>` root element:

```
  <item
    android:id="@+id/settings"
    android:icon="@drawable/ic_settings_black_24dp"
    android:showAsAction="never"
    android:title="@string/settings">
  </item>
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/res/menu/options.xml)

Step #4: Launching the Preference Activity

The only thing yet needed to allow the user to get to the preferences is to add another `case` to the `switch()` statement in `onOptionsItemSelected()` of `EmPubLiteActivity`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.about:
        Intent i = new Intent(this, SimpleContentActivity.class)
          .putExtra(SimpleContentActivity.EXTRA_FILE,
```

```
          "file:///android_asset/misc/about.html");
      startActivity(i);

      return(true);

    case R.id.help:
      i = new Intent(this, SimpleContentActivity.class)
        .putExtra(SimpleContentActivity.EXTRA_FILE,
          "file:///android_asset/misc/help.html");
      startActivity(i);

      return(true);

    case R.id.settings:
      startActivity(new Intent(this, Preferences.class));

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Now, if you run this in an emulator or device, you will see the new option in the action bar overflow:

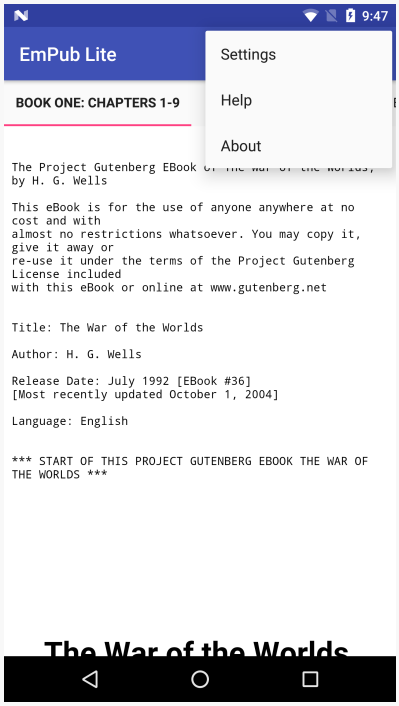


Figure 292: EmPubLite, With Revised Action Bar

Choosing the “Settings” option brings up our two preferences:

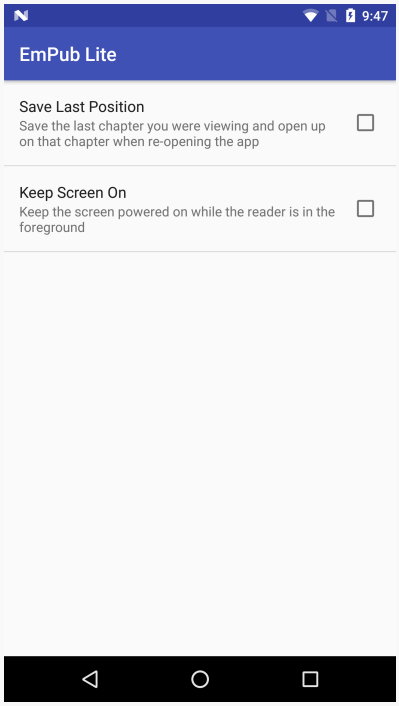


Figure 293: Our Preferences

Step #5: Loading the Preferences

Now, we need to actually arrange to load the preferences on a background thread. As noted, this will be handled by our `ModelFragment`, much as it handles the loading of the book contents.

First, add a private data member named `prefs`, that is an `AtomicReference` to a `SharedPreferences`, to `ModelFragment`:

```
final private AtomicReference<SharedPreferences> prefs=
  new AtomicReference<>();
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

Then, add a `getPrefs()` method to `ModelFragment` that returns the `prefs` value:

```
public SharedPreferences getPrefs() {
  return(prefs.get());
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

Next, revise `LoadThread` to:

- Replace the `assets` data member with a `ctxt` data member of type `Context`
- Take in a `Context` in the constructor, instead of an `AssetManager` (this way, even if for some strange reason our original activity is destroyed and recreated while we are loading the preferences, we will not be leaking the original activity)
- Save the application context (from `getApplicationContext()` on `Context`) in a data member, instead of an `AssetManager`
- Call `getAssets()` on that `Context` in `run()`, instead of using the former `AssetManager`
- Also retrieve the `SharedPreferences` in `run()`

```
private class LoadThread extends Thread {
  final private Context ctxt;

  LoadThread(Context ctxt) {
    super();

    this.ctxt=ctxt.getApplicationContext();
  }

  @Override
  public void run() {
    prefs.set(PreferenceManager.getDefaultSharedPreferences(ctxt));

    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    Gson gson=new Gson();

    try {
      InputStream is=ctxt.getAssets().open("book/contents.json");
      BufferedReader reader=
        new BufferedReader(new InputStreamReader(is));

      contents.set(gson.fromJson(reader, BookContents.class));

      EventBus.getDefault().post(new BookLoadedEvent(getBook()));
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

This has our `LoadThread` load both the `SharedPreferences` and the `BookContents`, and do so in a known order (`SharedPreferences` first).

You will need to modify `onAttach()` to just pass in the `Activity` to the `LoadThread` constructor:

```
  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    if (contents.get()==null) {
      new LoadThread(host).start();
    }
  }
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

The resulting `ModelFragment` should look like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.content.Context;
import android.content.SharedPreferences;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.os.Process;
import android.preference.PreferenceManager;
import android.util.Log;
import com.google.gson.Gson;
import org.greenrobot.eventbus.EventBus;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.concurrent.atomic.AtomicReference;

public class ModelFragment extends Fragment {
  final private AtomicReference<BookContents> contents=
    new AtomicReference<>();
  final private AtomicReference<SharedPreferences> prefs=
    new AtomicReference<>();

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setRetainInstance(true);
  }

  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    if (contents.get()==null) {
      new LoadThread(host).start();
    }
  }

  public BookContents getBook() {
    return(contents.get());
  }

  public SharedPreferences getPrefs() {
    return(prefs.get());
  }

  private class LoadThread extends Thread {
    final private Context ctxt;

    LoadThread(Context ctxt) {
      super();

      this.ctxt=ctxt.getApplicationContext();
    }

    @Override
    public void run() {
      prefs.set(PreferenceManager.getDefaultSharedPreferences(ctxt));

      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
      Gson gson=new Gson();

      try {
        InputStream is=ctxt.getAssets().open("book/contents.json");
        BufferedReader reader=
          new BufferedReader(new InputStreamReader(is));

        contents.set(gson.fromJson(reader, BookContents.class));

        EventBus.getDefault().post(new BookLoadedEvent(getBook()));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
```

```
    }
  }
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

Step #6: Saving the Last-Read Position

One preference is to restore our current page in the `ViewPager` when the user later re-opens the app. To make that work, we need to start saving the current page as the user leaves the app. And, we may as well use our freshly-minted `SharedPreferences` to store this value.

We need a key under which we will store this value in the `SharedPreferences`, so add a new static data member to `EmPubLiteActivity`:

```
private static final String PREF_LAST_POSITION="lastPosition";
```

We are also going to need access to our `ModelFragment` from outside of `onResume()` in `EmPubLiteActivity`. Add a `ModelFragment` data member named `mfrag`:

```
  private ModelFragment mfrag=null;
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Then, modify `onStart()` to refer to the `mfrag` data member, replacing the former `mfrag` local variable:

```
  @Override
  public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);

    if (adapter==null) {
      mfrag=(ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

      if (mfrag==null) {
        mfrag=new ModelFragment();

        getFragmentManager().beginTransaction()
          .add(mfrag, MODEL).commit();
      }
      else if (mfrag.getBook()!=null) {
        setupPager(mfrag.getBook());
      }
```

```
    }
  }
```

Next, update `onStop()` on `EmPubLiteActivity` to track the page of the `ViewPager` that the user is on at the point in time when `onStop()` is called:

```
@Override
public void onStop() {
  EventBus.getDefault().unregister(this);

  if (mfrag.getPrefs()!=null) {
    int position=pager.getCurrentItem();

    mfrag.getPrefs().edit().putInt(PREF_LAST_POSITION, position)
      .apply();
  }

  super.onStop();
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Here, we check to see that we have the `SharedPreferences` loaded — odds are that we do, but we cannot be certain. If we do have access to the `SharedPreferences`, we find out the current position within the `ViewPager` via `getCurrentItem()` (e.g., `0` for the first page). We then obtain a `SharedPreferences.Editor` and use it to save this position value in the `SharedPreferences`, keyed as `PREF_LAST_POSITION`, using `apply()` to persist the changes.

Step #7: Restoring the Last-Read Position

Now that we are saving this position data, we can start to use it.

Our preference XML has our key to the "Save Last Position" preference, but we need it in Java code as well, so add another static data member to `EmPubLiteActivity`:

```
private static final String PREF_SAVE_LAST_POSITION="saveLastPosition";
```

Add the following lines to the end of `setupPager()` in `EmPubLiteActivity`:

```
    SharedPreferences prefs=mfrag.getPrefs();

    if (prefs != null) {
      if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
```

```
      pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
    }
  }
```

Here, we check to see if the user has enabled having us restore the last-saved position (defaulting to `false`). If the user has, we retrieve the last-saved position (defaulting to `0`, or the first page), and call `setCurrentItem()` on the `ViewPager` to shift to that particular page.

If you run this in a device or emulator, check the "Save Last Position" preference checkbox, flip ahead a couple of chapters, exit the app via the BACK button, and go back into the app, you will see that you are taken back to the chapter you were last reading.

Step #8: Keeping the Screen On

Our other preference is whether or not the screen should stay on, without user input, while we are reading the book. The bare-bones implementation of this requires just two lines of additional code.

First, we need to define another static data member on `EmPubLiteActivity`, this time with the key for our keep-screen-on preference:

```
private static final String PREF_KEEP_SCREEN_ON="keepScreenOn";
```

Then, add one more line to `setupPager()` in `EmPubLiteActivity`, inside of the `if` block:

```
pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
```

This will give you:

```
  private void setupPager(BookContents contents) {
    adapter=new ContentsAdapter(this, contents);
    pager.setAdapter(adapter);

    MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);

    SharedPreferences prefs=mfrag.getPrefs();

    if (prefs!=null) {
      if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
```

```
      pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
    }

    pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
  }
}
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

`setKeepScreenOn()`, called on any `View`, will keep the screen lit and active without continuous user input, so long as that `View` is on the screen.

This approach is somewhat limited, in that we are only setting this during the call to `setupPager()`. If the user changes the preference value, that change would only take effect when the activity was restarted (e.g., user rotates the screen, user exits the app via BACK and returns later).

The simplest way for us to have this take more immediate effect is to realize that `EmPubLiteActivity` will be paused and stopped when the `Preferences` activity is on the screen, and will be started and resumed when the user is done adjusting preferences. So, we can simply augment `onStart()` to also update the screen-on setting:

```
@Override
public void onStart() {
  super.onStart();
  EventBus.getDefault().register(this);

  if (adapter==null) {
    mfrag=(ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (mfrag==null) {
      mfrag=new ModelFragment();

      getFragmentManager().beginTransaction()
        .add(mfrag, MODEL).commit();
    }
    else if (mfrag.getBook()!=null) {
      setupPager(mfrag.getBook());
    }
  }

  if (mfrag.getPrefs()!=null) {
    pager.setKeepScreenOn(mfrag.getPrefs()
      .getBoolean(PREF_KEEP_SCREEN_ON, false));
```

```
    }
  }
```

(from EmPubLite-AndroidStudio/T13-Prefs/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Of course, we may not *have* the `SharedPreferences` yet, when the app is first starting up, so we avoid making any changes in that case.

If you run this on a device (note: *not* an emulator), you can play with this preference and see the changes in the screen's behavior.

In Our Next Episode…

… we will allow the user to write, save, and delete notes for the currently-viewed chapter, using a database.

SQLite Databases

Besides `SharedPreferences` and your own file structures, the third primary means of persisting data locally on Android is via SQLite. For many applications, SQLite is the app's backbone, whether it is used directly or via some third-party wrapper.

This chapter will focus on how you can directly work with SQLite to store relational data.

Introducing SQLite

SQLite is a very popular embedded database, as it combines a clean SQL interface with a very small memory footprint and decent speed. Moreover, it is public domain, so everyone can use it. Lots of firms (Adobe, Apple, Google, Symbian) and open source projects (Mozilla, PHP, Python) all ship products with SQLite.

For Android, SQLite is "baked into" the Android runtime, so every Android application can create SQLite databases. Since SQLite uses a SQL interface, it is fairly straightforward to use for people with experience in other SQL-based databases. However, its native API is not JDBC, and JDBC might be too much overhead for a memory-limited device like a phone, anyway. Hence, Android programmers have a different API to learn — the good news being is that it is not that difficult.

This chapter will cover the basics of SQLite use in the context of working on Android. It by no means is a thorough coverage of SQLite as a whole. If you want to learn more about SQLite, the SQLite Web site may help.

Thinking About Schemas

SQLite is a typical relational database, containing tables (themselves consisting of rows and columns), indexes, and so on. Your application will need its own set of tables and so forth for holding whatever data you wish to hold. This structure is generally referred to as a "schema".

It is likely that your schema will need to change over time. You might add new tables or columns in support of new features. Or, you might significantly reorganize your data structure and wind up dropping some tables while moving the data into new ones.

As a result, when you ship an update to your application to your users, not only will your Java code change, but the *expectations* of that Java code will change as well, with respect to what your database schema will look like. Version 1 of your app will use your original schema, but by the time you ship, say, version 5 of the app, you might need an adjusted schema.

Android has facilities to assist you with handling changing database schemas, mostly centered around the `SQLiteOpenHelper` class.

Start with a Helper

`SQLiteOpenHelper` is designed to consolidate your code related to two very common problems:

1. What happens the very first time when your app is run on a device after it is installed? At this point, we do not yet have a database, and so you will need to create your tables, indexes, starter data, and so on.
2. What happens the very first time when an upgraded version of your app is run on a device, where the upgraded version is expecting a newer database schema? Your database will still be on the old schema from the older edition of the app. You will need to have a chance to alter the database schema to match the needs of the rest of your app.

`SQLiteOpenHelper` wraps up the logic to create and upgrade a database, per your specifications, as needed by your application. You will need to create a custom subclass of `SQLiteOpenHelper`, implementing three methods at minimum:

1. The constructor, chaining upward to the `SQLiteOpenHelper` constructor. This takes the `Context` (e.g., an `Activity`), the name of the database, an optional cursor factory (typically, just pass `null`), and an integer representing the version of the database schema you are using (typically start at `1` and increment from there).
2. `onCreate()`, called when there is no database and your app needs one, which passes you a `SQLiteDatabase` object, pointing at a newly-created database, that you use to populate with tables and initial data, as appropriate.
3. `onUpgrade()`, called when the schema version you are seeking does not match the schema version of the database, which passes you a `SQLiteDatabase` object and the old and new version numbers, so you can figure out how best to convert the database from the old schema to the new one.

To see how all this SQLite stuff works in practice, we will examine the Database/ConstantsROWID sample application. This application pulls a bunch of gravitational constants from the `SensorManager` class, puts them in a database table, displays them in a `ListFragment`, and allows the user to add new ones via the action bar.

First, we need a `SQLiteOpenHelper` subclass, here named `DatabaseHelper`.

The `DatabaseHelper` constructor chains to the superclass and supplies the name of the database (held in a `DATABASE_NAME` static data member) and the version number of our database schema (held in `SCHEMA`):

```
public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";
  private static final int SCHEMA=1;
  static final String TITLE="title";
  static final String VALUE="value";
  static final String TABLE="constants";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, SCHEMA);
  }
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

We also need an `onCreate()` method, which will be called and passed a `SQLiteDatabase` object when a database needs to be newly created. Below you will see the `DatabaseHelper` implementation of `onCreate()`, though we will get into how it is using the `SQLiteDatabase` object more later in this chapter:

```
@Override
public void onCreate(SQLiteDatabase db) {
  db.execSQL("CREATE TABLE constants (title TEXT, value REAL);");

  ContentValues cv=new ContentValues();

  cv.put(TITLE, "Gravity, Death Star I");
  cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Earth");
  cv.put(VALUE, SensorManager.GRAVITY_EARTH);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Jupiter");
  cv.put(VALUE, SensorManager.GRAVITY_JUPITER);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Mars");
  cv.put(VALUE, SensorManager.GRAVITY_MARS);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Mercury");
  cv.put(VALUE, SensorManager.GRAVITY_MERCURY);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Moon");
  cv.put(VALUE, SensorManager.GRAVITY_MOON);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Neptune");
  cv.put(VALUE, SensorManager.GRAVITY_NEPTUNE);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Pluto");
  cv.put(VALUE, SensorManager.GRAVITY_PLUTO);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Saturn");
  cv.put(VALUE, SensorManager.GRAVITY_SATURN);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, Sun");
  cv.put(VALUE, SensorManager.GRAVITY_SUN);
  db.insert(TABLE, TITLE, cv);

  cv.put(TITLE, "Gravity, The Island");
  cv.put(VALUE, SensorManager.GRAVITY_THE_ISLAND);
```

```
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Uranus");
    cv.put(VALUE, SensorManager.GRAVITY_URANUS);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Venus");
    cv.put(VALUE, SensorManager.GRAVITY_VENUS);
    db.insert(TABLE, TITLE, cv);
  }
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

Suffice it to say for the moment that it is creating a `constants` table and inserting several rows into it, all wrapped in a transaction.

We also need `onUpgrade()`... even though it should never be called right now:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                      int newVersion) {
  throw new RuntimeException("How did we get here?");
}
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

After all, right now, we only have one version of our schema (`1`) and therefore will have no need to upgrade. If, in the future, we change `SCHEMA` to a higher value (e.g., `2`), and we upgrade our app on a device that had previously been run with our earlier schema, *then* we will be called with `onUpgrade()`. We are passed the old and new schema versions, so we know what needs to be upgraded.

Bear in mind that users do not necessarily have to take on each of your application updates, and so you might find that a user skipped a schema version:

- You release an app on Monday, with schema version `1`
- A user installs your app on Tuesday and runs it, creating a database via `onCreate()`
- You release an upgraded app on Wednesday, with schema version `2`
- You release yet another upgrade on Thursday, with schema version `3`
- The user installs your upgrade, now needing a schema version `3` database instead of the version `1` presently on the device, triggering a call to `onUpgrade()`

There are two other methods you can elect to override in your `SQLiteOpenHelper`, if you feel the need:

- You can override `onOpen()`, to get control when somebody opens this database. Usually, this is not required.
- Android 3.0 introduced `onDowngrade()`, which will be called if the code requests an older schema than what is in the database presently. This is the converse of `onUpgrade()` — if your version numbers differ, one of these two methods will be invoked. Since normally you are moving forward with updates, you can usually skip `onDowngrade()`.

Employing Your Helper

To use your `SQLiteOpenHelper` subclass, create and hold onto an instance of it. Then, when you need a `SQLiteDatabase` object to do queries or data modifications, ask your `SQLiteOpenHelper` to `getReadableDatabase()` or `getWritableDatabase()`, depending upon whether or not you will be changing its contents.

For example, the `ConstantsFragment` from the sample app creates a `DatabaseHelper` instance in `onViewCreated()` and holds onto it in a data member:

```
      db=new DatabaseHelper(getActivity());
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

When you are done with the database (e.g., your activity is being closed), simply call `close()` on your `SQLiteOpenHelper` to release your connection, as `ConstantsFragment` does (among other things) in `onDestroy()`:

```
@Override
public void onDestroy() {
  if (task!=null) {
    task.cancel(false);
  }

  ((CursorAdapter)getListAdapter()).getCursor().close();
  db.close();

  super.onDestroy();
}
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

(we will explore those “other things” in a bit)

Where to Hold a Helper

For trivial apps, like the one profiled in this chapter, holding a `SQLiteOpenHelper` in a data member of your one-and-only activity is fine.

If, however, you have multiple components — such as multiple activities – all needing to use the database, you are much better served having a *singleton instance* of your `SQLiteOpenHelper`, compared to having each activity have its own instance.

The reason is threading.

You really should do your database I/O on background threads. Opening a database is cheap, but working with it (queries, inserts, etc.) is not. The `SQLiteDatabase` object managed by `SQLiteOpenHelper` is thread-safe... so long as all threads are using the same instance.

For singleton objects that depend upon a `Context`, like `SQLiteOpenHelper`, rather than create the object using a garden-variety `Context` like an `Activity`, you really should create it with an `Application`. There is a singleton instance of a `Context`, in the form of the `Application` subclass, created in your process moments after it is started. You can retrieve this singleton by calling `getApplicationContext()` on any other `Context`. The advantage of using `Application` is memory leaks: if you put a `SQLiteOpenHelper` in a singleton, and use, say, an `Activity` to create it, then the `Activity` might not be able to be garbage-collected, because the `SQLiteOpenHelper` keeps a strong reference to it. Since `Application` is itself a singleton (and, hence, is "pre-leaked", so to speak), the risks of a memory leak diminish significantly.

So, instead of:

```
db=new DatabaseHelper(getActivity());
```

in a fragment, with `db` as a data member, you might have:

```
db=new DatabaseHelper(getActivity().getApplicationContext());
```

with `db` as a static data member, shared by multiple activities or other components.

Getting Data Out

One popular thing to do with a database is to get data out of it. Android has a few ways you can execute a query on a `SQLiteDatabase` (from your `SQLiteOpenHelper`),

along with some classes, like `CursorAdapter`, to help you use the results you get back.

Your Query Options

In most cases, your simplest option for executing a query is to call `rawQuery()` on the `SQLiteDatabase`. This takes two parameters:

- A SQL `SELECT` statement (or anything else that returns a result set), optionally with `?` characters in the `WHERE` clause (or `ORDER BY` or similar clauses) representing parameters to be bound at runtime
- An optional `String` array of the parameters to be used to replace the `?` characters in the query

If you do not use the `?` position parameter syntax in your query, you are welcome to pass `null` as the second parameter to `rawQuery()`.

The nice thing about `rawQuery()` is that any valid SQL syntax works, so long as it returns a result set. You are welcome to use joins, sub-selects, and so on without issue.

There are two other query options — `query()` and `SQLiteQueryBuilder`. These both build up a SQL `SELECT` statement from its component parts (e.g., name of the table to query, `WHERE` clause and positional parameters). These are more cumbersome to use, particularly with complex `SELECT` statements. Mostly, they would be used in cases where, for one reason or another, you do not know the precise query at compile time and find it easier to use these facilities to construct the query from parts at runtime. Some developers will do this to avoid duplicating values, by defining constants for things like table names and column names.

For example, `ConstantsFragment` has a private inner class named `BaseTask` which has a `doQuery()` method that uses `query()`:

```
abstract private class BaseTask<T> extends AsyncTask<T, Void, Cursor> {
  @Override
  public void onPostExecute(Cursor result) {
    ((CursorAdapter)getListAdapter()).changeCursor(result);
    current=result;
    task=null;
  }

  Cursor doQuery() {
```

```
    Cursor result=
        db
            .getReadableDatabase()
            .query(DatabaseHelper.TABLE,
                new String[] {"ROWID AS _id",
                              DatabaseHelper.TITLE,
                              DatabaseHelper.VALUE},
                null, null, null, null, DatabaseHelper.TITLE);

    result.getCount();

    return(result);
  }
}
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

Do not concatenate your own `WHERE` clause, though. Let the `?` positional parameters handle that for you, as the work they do to escape your apostrophes, quotation marks, and the like also helps to defend against SQL injection attacks. In this particular case, we do not have a `WHERE` clause.

If that `ROWID AS _id` piece looks a bit odd, we will see why that is in the query a bit later in this chapter.

What Is a Cursor?

All three of these give you a `Cursor` when you are done. In Android, a `Cursor` represents the entire result set of the query — all the rows and all the columns that the query returned. In this respect, it is reminiscent of a "client-side cursor" from toolkits like ODBC, JDBC, etc.

(if the `Cursor` result set is over 1MB, it actually only holds a "window" on the data, and the story gets really really complicated...)

As such, a `Cursor` can be quite the memory hog. Please `close()` the `Cursor` when you are done with it, to free up the heap space it consumes and make that memory available to the rest of your application.

Using the Cursor Manually

With the `Cursor`, you can:

1. Find out how many rows are in the result set via `getCount()`
2. Iterate over the rows via `moveToFirst()`, `moveToNext()`, and `isAfterLast()`
3. Find out the names of the columns via `getColumnNames()`, convert those into column numbers via `getColumnIndex()`, and get values for the current row for a given column via methods like `getString()`, `getInt()`, etc.

For example, here we iterate over a fictitious `widgets` table's rows:

```
Cursor result=
  db.rawQuery("SELECT _id, name, inventory FROM widgets", null);

while (result.moveToNext()) {
  int id=result.getInt(0);
  String name=result.getString(1);
  int inventory=result.getInt(2);

  // do something useful with these
}

result.close();
```

Introducing CursorAdapter

Another way to use a `Cursor` is to wrap it in a `CursorAdapter`. Just as `ArrayAdapter` adapts arrays, `CursorAdapter` adapts `Cursor` objects, making their data available to an `AdapterView` like a `ListView`.

The easiest way to set one of these up is to use `SimpleCursorAdapter`, which extends `CursorAdapter` and provides some boilerplate logic for taking values out of columns and putting them into row `View` objects for a `ListView` (or other `AdapterView`). The sample app does just that:

```
    SimpleCursorAdapter adapter=
        new SimpleCursorAdapter(getActivity(), R.layout.row,
            current, new String[] {
            DatabaseHelper.TITLE,
            DatabaseHelper.VALUE },
            new int[] { R.id.title, R.id.value },
            0);

    setListAdapter(adapter);
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

Here, we are telling `SimpleCursorAdapter` to take rows out of a `Cursor` named `current`, turning each into an inflated `R.layout.row` `ViewGroup`, in this case, a `RelativeLayout` holding a pair of `TextView` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:textSize="20sp"
    android:textStyle="bold"/>

  <TextView
    android:id="@+id/value"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:textSize="20sp"
    android:textStyle="bold"/>

</RelativeLayout>
```

(from Database/ConstantsROWID/app/src/main/res/layout/row.xml)

For each row in the `Cursor`, the columns named `title` and `value` (represented by `TITLE` and `VALUE` constants on `DatabaseHelper`) are to be poured into their respective `TextView` widgets (`R.id.title` and `R.id.value`).

Note, though, that if you are going to use `CursorAdapter` or its subclasses (like `SimpleCursorAdapter`), your result set of your query *must* contain an integer column named `_id` that is unique for the result set. This "id" value is then supplied to methods like `onListItemClick()`, to identify what item the user clicked upon in the `AdapterView`. Note that this requirement is on the result set in the `Cursor`, so if you have a suitable column in a table that is not named `_id`, you can rename it in your query (e.g., `SELECT key AS _id, ...`).

However, if you want, you can use the built-in `ROWID`

Quoting the SQLite documentation:

> In SQLite, every row of every table has a 64-bit signed integer ROWID. The ROWID for each row is unique among all rows in the same table. You can access the ROWID of an SQLite table using one the special column names ROWID, _ROWID_, or OID... If a table contains a column of type INTEGER PRIMARY KEY, then that column becomes an alias for the ROWID. You can then access the ROWID using any of four different names, the original three names described above or the name given to the INTEGER PRIMARY KEY column. All these names are aliases for one another and work equally well in any context.

With that in mind, if you want to query SQLite and use the results in a `CursorAdapter`, but you do not have your own `INTEGER PRIMARY KEY` column, you can just include `ROWID` in your query, renaming it to `_id` to satisfy `CursorAdapter`.

That is why we have the `ROWID AS _id` in the `doQuery()` method: to satisfy this `_id` requirement of `CursorAdapter`.

Also note that you cannot `close()` the `Cursor` used by a `CursorAdapter` until you no longer need the `CursorAdapter`. That is why we do not close the `Cursor` until `onDestroy()` of the fragment:

```
@Override
public void onDestroy() {
  if (task!=null) {
    task.cancel(false);
  }

  ((CursorAdapter)getListAdapter()).getCursor().close();
  db.close();

  super.onDestroy();
}
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

We retrieve the `Cursor` from the `CursorAdapter`, which we get by calling `getListAdapter()` on the fragment.

Getting Data Out, Asynchronously

Ideally, queries are done on a background thread, as they may take some time.

One approach for doing that is to use an `AsyncTask`. In the sample application, `ConstantsFragment` kicks off a `LoadCursorTask` in `onViewCreated()` (shown above). `LoadCursorTask` extends the `BaseTask` class mentioned previously, where the `doQuery()` method resides. `LoadCursorTask` is responsible for doing the query (via the `doQuery()` method shown above) and putting the results in the `ListView` inside the fragment (using the `SimpleCursorAdapter` shown above):

```
abstract private class BaseTask<T> extends AsyncTask<T, Void, Cursor> {
  @Override
  public void onPostExecute(Cursor result) {
    ((CursorAdapter)getListAdapter()).changeCursor(result);
    current=result;
    task=null;
  }

  Cursor doQuery() {
    Cursor result=
        db
            .getReadableDatabase()
            .query(DatabaseHelper.TABLE,
                new String[] {"ROWID AS _id",
                              DatabaseHelper.TITLE,
                              DatabaseHelper.VALUE},
                null, null, null, null, DatabaseHelper.TITLE);

    result.getCount();

    return(result);
  }
}

private class LoadCursorTask extends BaseTask<Void> {
  @Override
  protected Cursor doInBackground(Void... params) {
    return(doQuery());
  }
}
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

We execute the actual query in `doInBackground()` and call `getCount()` on the `Cursor`, to force it to actually perform the query — `query()` returns the `Cursor`, but the query is not actually executed until we do something that needs the result set. This also holds true for `rawQuery()`, which is why we need to make sure to "touch" the `Cursor` while we are on the background thread.

onPostExecute() then uses changeCursor() to replace the Cursor in the SimpleCursorAdapter with the results. Since our SimpleCursorAdapter was created with a null Cursor, changeCursor() just slides in the new Cursor, telling the ListView that the data changed. This causes our ListView to be populated.

This way, the UI will not be frozen while the query is being executed, yet we only update the UI from the main application thread.

Note that the first time we try using the SQLiteOpenHelper is in our background thread. SQLiteOpenHelper will not try creating our database (e.g., for a new app install) until we call getReadableDatabase() or getWritableDatabase(). Hence, onCreate() (or, later, onUpgrade()) of our SQLiteOpenHelper will wind up being called on the background thread as well, meaning that the time spent creating (or upgrading) the database also does not freeze the UI.

Also note that in onDestroy(), as shown previously, we call cancel() on the AsyncTask if it is not null. If the task is still running, calling cancel() will prevent onPostExecute() from being invoked, and we will not have to worry about updating our UI after the fragment has been destroyed.

The Rest of the CRUD

To get data out of a database, it is generally useful to put data into it in the first place. The sample app starts by loading in data when the database is created (in onCreate() of DatabaseHelper), plus has an action bar item to allow the user to add other constants as needed.

In this section, we will examine in further detail how we manipulate the database, for both the write aspects of CRUD (create-read-update-delete) and for data definition language (DDL) operations (creating tables, creating indexes, etc.).

The Primary Option: execSQL()

For creating your tables and indexes, you will need to call execSQL() on your SQLiteDatabase, providing the DDL statement you wish to apply against the database. Barring a database error, this method returns nothing.

So, for example, you can call execSQL() to create the constants table, as shown in the DatabaseHelper onCreate() method:

```
    db.execSQL("CREATE TABLE constants (title TEXT, value REAL);");
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

This will create a table, named `constants`, with two data columns: `title` (text) and `value` (a float, or "real" in SQLite terms).

Most likely, you will create tables and indexes when you first create the database, or possibly when the database needs upgrading to accommodate a new release of your application. If you do not change your table schemas, you might never drop your tables or indexes, but if you do, just use `execSQL()` to invoke `DROP INDEX` and `DROP TABLE` statements as needed.

Alternative Options

For inserts, updates, and deletes of data, you have two choices. You can always use `execSQL()`, just like you did for creating the tables. The `execSQL()` method works for any SQL that does not return results, so it can handle `INSERT`, `UPDATE`, `DELETE`, etc. just fine.

Your alternative is to use the `insert()`, `update()`, and `delete()` methods on the `SQLiteDatabase` object, which eliminate much of the SQL syntax required to do basic operations.

For example, here we `insert()` a new row into our `constants` table, again from `onCreate()` of `DatabaseHelper`:

```
    ContentValues cv=new ContentValues();

    cv.put(TITLE, "Gravity, Death Star I");
    cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
    db.insert(TABLE, TITLE, cv);
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

These methods make use of `ContentValues` objects, which implement a `Map`-esque interface, albeit one that has additional methods for working with SQLite types. For example, in addition to `get()` to retrieve a value by its key, you have `getAsInteger()`, `getAsString()`, and so forth.

The `insert()` method takes the name of the table, the name of one column as the "null column hack", and a `ContentValues` with the initial values you want put into this row. The "null column hack" is for the case where the `ContentValues` instance is

empty — the column named as the "null column hack" will be explicitly assigned the value `NULL` in the SQL `INSERT` statement generated by `insert()`. This is required due to a quirk in SQLite's support for the SQL `INSERT` statement.

The `update()` method takes the name of the table, a `ContentValues` representing the columns and replacement values to use, an optional `WHERE` clause, and an optional list of parameters to fill into the `WHERE` clause, to replace any embedded question marks (`?`). Since `update()` only replaces columns with fixed values, versus ones computed based on other information, you may need to use `execSQL()` to accomplish some ends. The `WHERE` clause and parameter list works akin to the positional SQL parameters you may be used to from other SQL APIs.

The `delete()` method works akin to `update()`, taking the name of the table, the optional `WHERE` clause, and the corresponding parameters to fill into the `WHERE` clause.

Asynchronous CRUD and UI Updates

Just as querying a database should be done on a background thread, so should modifying a database. This is why it is important to make the first time you request a `SQLiteDatabase` from a `SQLiteOpenHelper` be on a background thread, in case `onCreate()` or `onUpgrade()` are needed.

The same thing holds true if you need to update the database during normal operation of your app. For example, the sample application has an "add" action bar item in the upper-right corner of the screen:

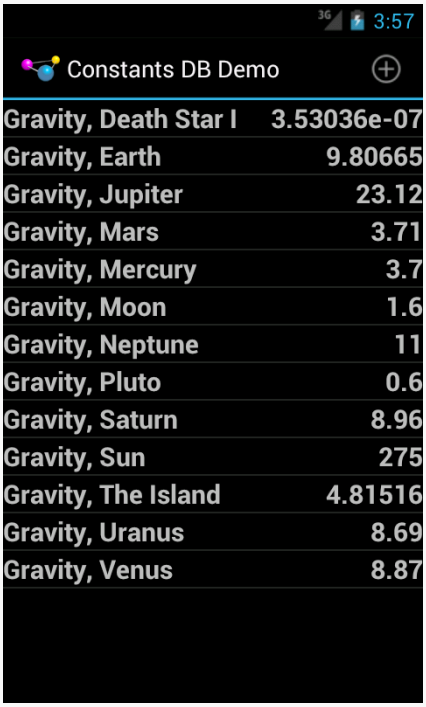


Figure 294: The ConstantsBrowser Sample

Clicking on that brings up a dialog — a technique we will discuss later in this book:

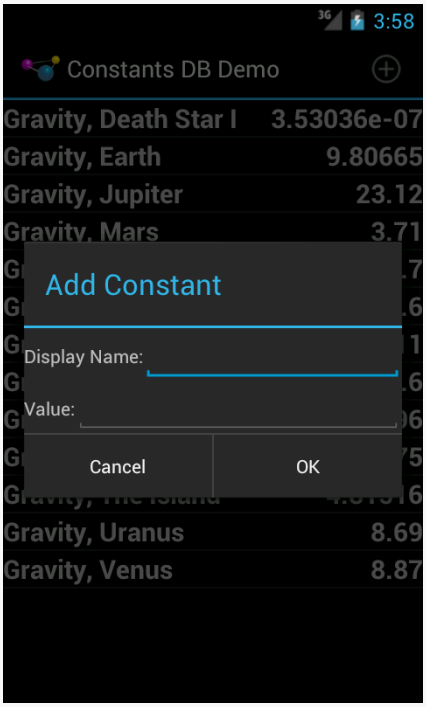


Figure 295: The ConstantsBrowser Sample, Add Constant Dialog

If the user fills in a constant and clicks the "OK" button, we need to insert a new record in the database. That is handled via an `InsertTask`:

```
private class InsertTask extends BaseTask<ContentValues> {
  @Override
  protected Cursor doInBackground(ContentValues... values) {
    db.getWritableDatabase().insert(DatabaseHelper.TABLE,
                                    DatabaseHelper.TITLE, values[0]);

    return(doQuery());
  }
}
```

(from Database/ConstantsROWID/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

The `InsertTask` is supplied a `ContentValues` object with our `title` and `value`, just as we used in `onCreate()` of `DatabaseHelper`. In `doInBackground()`, we get a writable database from `DatabaseHelper` and perform the `insert()` call, so the database I/O does not tie up the main application thread.

However, in `doInBackground()`, we *also* call `doQuery()` again. This retrieves a fresh `Cursor` with the new roster of constants... including the one we just inserted. As with

LoadCursorTask, we execute doQuery() in doInBackground() to keep the database I/O off the main application thread. This triggers the same onPostExecute() as before, inherited from BaseTask, which uses changeCursor() to replace any existing results with the new results.

Setting Transaction Bounds

By default, each SQL statement executes in its own transaction — this is fairly typical behavior for a SQL database, and SQLite is no exception.

There are two reasons why you might want to have your own transaction bounds, larger than a single statement:

1. The classic "we have a series of operations that need to succeed or fail as a whole" rationale, for maintaining data integrity
2. Performance, as each database transaction involves disk I/O, and one large transaction will be much faster than lots of little transactions

The basic recipe for your own transactions is:

```
try {
  db.beginTransaction();

  // several SQL statements in here

  db.setTransactionSuccessful();
}
finally {
  db.endTransaction();
}
```

beginTransaction() marks the fact that you want a transaction. setTransactionSuccessful() indicates that you want the transaction to commit. However, the actual COMMIT or ROLLBACK does not occur until endTransaction(). In the normal case, setTransactionSuccessful() does get called, and endTransaction() performs a COMMIT. If, however, one of your SQL statements fails (e.g., violates a foreign key constraint), the setTransactionSuccessful() call is skipped, so endTransaction() will do a ROLLBACK.

You might wonder why we did not bother with a transaction in onCreate() method of DatabaseHelper, given the latter reason. That is because onCreate() is called

within a transaction set up by `SQLiteOpenHelper` itself, so you do not need your own.

Hey, What About Hibernate?

Those of you with significant Java backgrounds outside of Android are probably pounding your head against your desk right about now. Outside of a few conveniences like `SQLiteOpenHelper` and `CursorAdapter`, Android's approach to database I/O feels a bit like classic JDBC. Java developers, having experienced the pain of raw JDBC, created various wrappers around it, the most prominent of which is an ORM (object-relational mapper) called Hibernate.

Alas, Hibernate is designed for servers, not mobile devices. It is a little bit heavyweight, and it is designed for use with JDBC, not Android's SQLite classes.

Android did not include any sort of ORM in the beginning for two main reasons:

1. To keep the firmware size as small as possible, as smaller firmware can lead to less-expensive devices
2. To eliminate the ORM overhead (e.g., reflection), which would have been too much for early Android versions on early Android devices

The Android ecosystem has come up with alternatives, such as ORMLite and greenDAO. So, if you are used to using an ORM, you may want to investigate these sorts of solutions — they just are not built into Android itself.

But, What About Room?

In 2017, Google introduced the Architecture Components: a set of libraries designed to offer higher-level abstractions around core architecture concerns. One piece of the Architecture Components is Room, which is Google's take on object-relational mapping code.

Room is covered in the companion volume, "Android's Architecture Components".

Visit the Trails!

If you are interested in exposing your database contents to a third-party application, you may wish to read up on `ContentProvider`.

The trails also have chapters on encrypted databases using SQLCipher and shipping pre-packaged databases with your app.

Tutorial #14 - Saving Notes

It would be nice if the user could add some personal notes to the chapter that she is reading, whether that serves as commentary, points to be researched, complaints about the author's hair (or lack thereof), or whatever.

So, in this chapter, we will add a new fragment and new activity to allow the user to add notes per chapter, via a large `EditText` widget. Those notes will be stored in a SQLite database.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Adding a DatabaseHelper

The first step for working with SQLite is to add an implementation of `SQLiteOpenHelper`, which we will do here, named `DatabaseHelper`.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `DatabaseHelper` as the name and `android.database.sqlite.SQLiteOpenHelper` as the superclass. Then, click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
```

```
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="empublite.db";
  private static final int SCHEMA_VERSION=1;
  private static DatabaseHelper singleton=null;

  synchronized static DatabaseHelper getInstance(Context ctxt) {
    if (singleton == null) {
      singleton=new DatabaseHelper(ctxt.getApplicationContext());
    }

    return(singleton);
  }

  private DatabaseHelper(Context ctxt) {
    super(ctxt, DATABASE_NAME, null, SCHEMA_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE notes (position INTEGER PRIMARY KEY, prose TEXT);");
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    throw new RuntimeException("This should not be called");
  }
}
```

Step #2: Examining DatabaseHelper

Our initial version of `DatabaseHelper` has a few things:

- It has the constructor, supplying to the superclass the name of the database file (`DATABASE_NAME`) and the revision number of our schema (`SCHEMA_VERSION`). Note that the constructor is private, as we are using the singleton pattern, so only `DatabaseHelper` should be able to create `DatabaseHelper` instances.
- It has the `onCreate()` method, invoked the first time we run the app on a device or emulator, to let us populate the database. Here, we use `execSQL()` to define a `notes` table with a `position` column (indicating our chapter) and a `prose` column (what the user types in as the note).

- It has the `onUpgrade()` method, needed because `SQLiteOpenHelper` is abstract, so our app will not compile without an implementation. Until we revise our schema, though, this method should never be called, so we raise a `RuntimeException` in the off chance that it *is* called unexpectedly.
- It has a static `DatabaseHelper` singleton instance and a `getInstance()` method to lazy-initialize it.

As noted in the chapter on databases, it is important to ensure that all threads are accessing the same `SQLiteDatabase` object, for thread safety. That usually means you hold onto a single `SQLiteOpenHelper` object. And, in our case, we might want to get at this database from more than one activity. Hence, we go with the singleton approach, so everyone works with the same `DatabaseHelper` instance.

Step #3: Creating a NoteFragment

Having a database is nice and all, but we need to work on the UI to allow users to enter notes. To do that, we will start with a `NoteFragment`.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `NoteFragment` as the name and `android.app.Fragment` as the superclass. Then, click OK to create the empty class.

Next, replace the contents of that class with the following:

```
package com.commonsware.empublite;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

public class NoteFragment extends Fragment {
  private static final String KEY_POSITION="position";
  private EditText editor=null;

  static NoteFragment newInstance(int position) {
    NoteFragment frag=new NoteFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
```

```
    frag.setArguments(args);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);

    editor=(EditText)result.findViewById(R.id.editor);

    return(result);
  }

  private int getPosition() {
    return(getArguments().getInt(KEY_POSITION, -1));
  }
}
```

Note that this fragment uses the `res/layout/editor.xml` resource that we created back in Tutorial #5.

Step #4: Examining NoteFragment

Our `NoteFragment` is fairly straightforward and is reminiscent of the `SimpleContentFragment` we created in Tutorial #11.

`NoteFragment` has a `newInstance()` static factory method. This method creates an instance of `NoteFragment`, takes a passed-in `int` (identifying the chapter for which we are creating a note), puts it in a `Bundle` identified as `KEY_POSITION`, hands the `Bundle` to the fragment as its arguments, and returns the newly-created `NoteFragment`.

In `onCreateView()`, we inflate the `R.layout.editor` resource that we defined and get our hands on our `EditText` widget for later use.

Step #5: Creating the NoteActivity

Having a fragment without displaying it is fairly pointless, so we need something to load a `NoteFragment`. Particularly for phones, the simplest answer is to create a

NoteActivity for that, paralleling the relationship between SimpleContentFragment and SimpleContentActivity.

Right-click over the com.commonsware.empublite package in your java/ directory and choose New > Activity > Empty Activity from the context menu. Fill in NoteActivity as the name, uncheck all the checkboxes, and click OK to create the empty class.

In the NoteActivity class that is created, replace the current implementation with the following:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;

public class NoteActivity extends Activity {
  public static final String EXTRA_POSITION="position";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      int position=getIntent().getIntExtra(EXTRA_POSITION, -1);

      if (position >= 0) {
        Fragment f=NoteFragment.newInstance(position);

        getFragmentManager().beginTransaction()
                            .add(android.R.id.content, f).commit();
      }
    }
  }
}
```

Step #6: Examining NoteActivity

As you can see, this is a fairly trivial activity. In onCreate(), if we are being created anew, we execute a FragmentTransaction to add a NoteFragment to our activity, pouring it into the full screen (android.R.id.content). Here, android.R.id.content identifies the container into which the results of

`setContentView()` would go — it is a container supplied by Activity itself and serves as the top-most container for our content.

However, we expect that we will be passed an `Intent` extra with the position (`EXTRA_POSITION`), which we pass along to the `NoteFragment` factory method.

Step #7: Add Notes to the Action Bar

Of course, none of this is useful if we do not give the user a way to get to the `NoteActivity`. Specifically, we can add a `notes` entry to our `res/menu/options.xml` resource, to have a new toolbar button appear on our main activity's action bar.

Right-click over the `res/` directory and choose New > Vector Asset from the context menu. Click the Icon button and search for the "Create" icon:

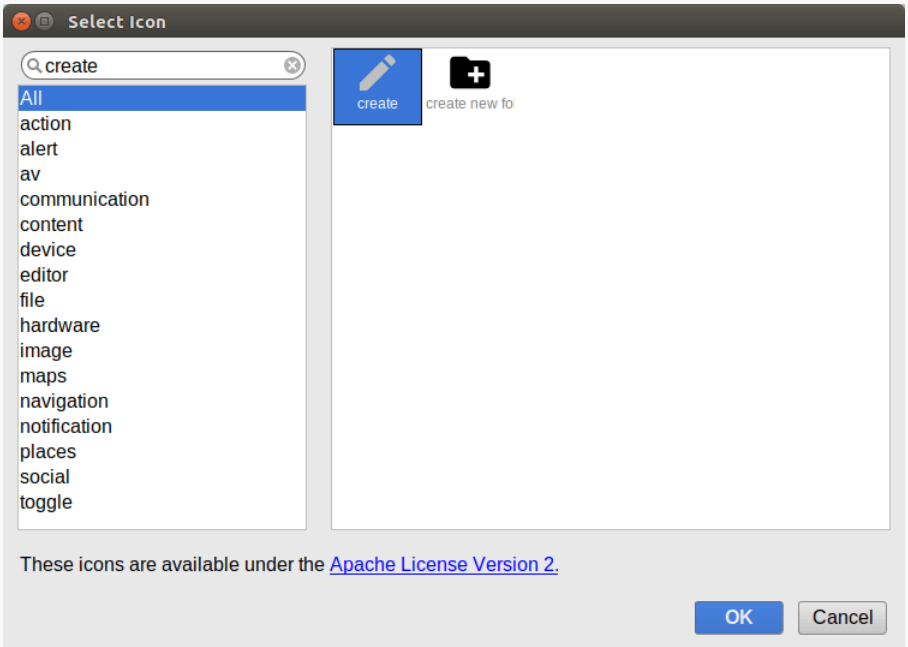


Figure 296: Asset Studio Icon Picker, with Create Icon Selected

Click OK to close the icon picker. Then click Next and Finish to save this drawable resource.

Unfortunately, this icon will render in black, when we need it to render in white given our theme. Right click over `res/drawable/ic_create_black_24dp.xml`, choose Refactor > Rename from the context menu, and change the name to

`ic_create_white_24dp.xml`. Then, open `res/drawable/ic_create_white_24dp.xml` and change the `android:fillColor` in the `<path>` element to be `#FFFFFFFF` instead of `#FF000000`:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
        android:width="24dp"
        android:height="24dp"
        android:viewportWidth="24.0"
        android:viewportHeight="24.0">
    <path
        android:fillColor="#FFFFFFFF"
        android:pathData="M3,17.25V21h3.75L17.81,9.94l-3.75,-3.75L3,17.25zM20.71,7.04c0.39,-0.39
0.39,-1.02 0,-1.41l-2.34,-2.34c-0.39,-0.39 -1.02,-0.39 -1.41,0l-1.83,1.83 3.75,3.75 1.83,-1.83z"/>
</vector>
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/res/drawable/ic_create_white_24dp.xml)

Next, add a new string resource, named `notes`, with a value like `Notes`.

Then, modify `res/menu/options.xml` to look like:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/notes"
    android:icon="@drawable/ic_create_white_24dp"
    android:showAsAction="ifRoom|withText"
    android:title="@string/notes">
  </item>
  <item
    android:id="@+id/settings"
    android:icon="@drawable/ic_settings_black_24dp"
    android:showAsAction="never"
    android:title="@string/settings">
  </item>
  <item
    android:id="@+id/help"
    android:icon="@drawable/ic_help_outline_black_24dp"
    android:title="@string/help">
  </item>
  <item
    android:id="@+id/about"
    android:icon="@drawable/ic_info_outline_black_24dp"
    android:title="@string/about">
  </item>

</menu>
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/res/menu/options.xml)

Finally, in `EmPubLiteActivity`, add the following `case` to the `switch` statement in `onOptionsItemSelected()`:

```
      case R.id.notes:
        startActivity(new Intent(this, NoteActivity.class)
          .putExtra(NoteActivity.EXTRA_POSITION,
            pager.getCurrentItem()));

        return(true);
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Here, we get the currently-viewed position from the `ViewPager` and pass that as the `EXTRA_POSITION` extra to `NoteActivity`.

Step #8: Defining a NoteLoadedEvent

We will want to load notes from the database on a background thread. Hence, we can apply the same basic approach as we used with `ModelFragment`, posting an event on the greenrobot EventBus when the load is completed, to deliver the results to the `NoteFragment`. This step will create a `NoteLoadedEvent` to handle this case.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `NoteLoadedEvent` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

class NoteLoadedEvent {
  int position;
  String prose;

  NoteLoadedEvent(int position, String prose) {
    this.position=position;
    this.prose=prose;
  }

  int getPosition() {
    return(position);
  }
```

```
  String getProse() {
    return(prose);
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteLoadedEvent.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #9: Loading a Note from the Database

Next, we need to add code somewhere that will actually query the database (on a background thread) to load the note for a given `ViewPager` position. One common pattern is to put this sort of database-access logic on your `SQLiteOpenHelper` subclass, so all of your database-specific code resides in one place. That is the approach we will take here, adding a `loadNote()` method that will fork a thread, query the database, and post a `NoteLoadedEvent` as a result.

Edit your `DatabaseHelper` to add its own `LoadThread` inner class, reminiscent of the one from `ModelFragment`:

```
private class LoadThread extends Thread {
  private int position=-1;

  LoadThread(int position) {
    super();
    this.position=position;
  }

  @Override
  public void run() {
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);

    String[] args={String.valueOf(position)};
    Cursor c=
      getReadableDatabase().rawQuery("SELECT prose FROM notes WHERE position = ? ", args);

    if (c.getCount() > 0) {
      c.moveToFirst();

      EventBus.getDefault().post(new NoteLoadedEvent(position,
        c.getString(0)));
    }

    c.close();
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/DatabaseHelper.java)

Here, we use `rawQuery()` to retrieve the note based upon a supplied position. If there is no such note, our `Cursor` will have no rows, and we are done. If, however, we did get results back on the query, we then post a `NoteLoadedEvent` with the `position` and the `prose` (the text from the database).

You will need to add an import manually to `android.os.Process`, to be able to resolve the `setThreadPriority()` method and its parameter.

Also, add a `loadNote()` method to `DatabaseHelper` that forks this `LoadThread`:

```
void loadNote(int position) {
  new LoadThread(position).start();
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/DatabaseHelper.java)

Step #10: Loading the Note Into the Fragment

Now that we can query the database and get back a note (if any), we can tie that into the `NoteFragment` to load the note for the fragment's `position` when the fragment is opened. We will not only need to call `loadNote()` on the `DatabaseHelper`, but also be able to respond to the `NoteLoadedEvent` when it arrives.

Add the following `onStart()` method to `NoteFragment`:

```
@Override
public void onStart() {
  super.onStart();

  EventBus.getDefault().register(this);

  if (TextUtils.isEmpty(editor.getText())) {
    DatabaseHelper db=DatabaseHelper.getInstance(getActivity());
    db.loadNote(getPosition());
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Here, we register for the `EventBus`. Then, if we do not have any text in the `EditText` widget, we call `loadNote()` on our singleton instance of the `DatabaseHelper`, passing in the `position` that our fragment is managing. The reason for checking to see if the `EditText` is empty is to handle configuration changes. This fragment is not a retained fragment, and so it will be destroyed and re-created. The default

`onSaveInstanceState()` logic of `EditText` will retain our note, though, so we do not want to re-load it from the database. This approach is not optimal, in that we will wind up calling `loadNote()` in cases where we could know that there is no note. That optimization is complex enough to not make it worthwhile for a set of book tutorials, though it is something you might wish to explore in a commercial-grade application.

Next, add the corresponding `onStop()` to `NoteFragment`, to unregister from the `EventBus`:

```
@Override
public void onStop() {
  EventBus.getDefault().unregister(this);

  super.onStop();
}
```

Finally, add an `onNoteLoaded(NoteLoadedEvent)` method to `NoteFragment`, so we receive the `NoteLoadedEvent` on the main application thread:

```
@SuppressWarnings("unused")
@Subscribe(threadMode =ThreadMode.MAIN)
public void onNoteLoaded(NoteLoadedEvent event) {
  if (event.getPosition() == getPosition()) {
    editor.setText(event.getProse());
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Here, we confirm that the event is for our fragment's position, as it is conceivable that this event is for some other note, though that is rather unlikely given how the user would view notes. That being said, if the note is for our position, we populate the `EditText` with the note prose.

Step #11: Updating the Database

Of course, loading notes from a database is all fine and well... except that we do not have any notes *in* the database. We really should fix that.

Add an `UpdateThread` inner class to `DatabaseHelper`:

```
private class UpdateThread extends Thread {
  private int position=-1;
```

```
  private String prose=null;

  UpdateThread(int position, String prose) {
    super();
    this.position=position;
    this.prose=prose;
  }

  @Override
  public void run() {
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);

    String[] args={String.valueOf(position), prose};
    getWritableDatabase().execSQL("INSERT OR REPLACE INTO notes (position, prose) VALUES (?, ?)",
      args);
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/DatabaseHelper.java)

Here, we use `execSQL()` to execute an `INSERT OR REPLACE` SQL statement. As the name suggests, this will insert a new row if there is no match on our primary key (`position`). Otherwise, it will update the other columns if there is a match.

Note that we do not post an event here. We could, if there was something in the app that needed to know when a note was updated.

Also, add an `updateNote()` method to `DatabaseHelper` that forks this `UpdateThread`:

```
void updateNote(int position, String prose) {
  new UpdateThread(position, prose).start();
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/DatabaseHelper.java)

Step #12: Saving the Note

Somewhere, we need to call `updateNote()`. A classic "desktop" approach would be to have a "save" action bar item in the `NoteFragment`, which the user would need to click upon to save the note. However, this does not deal with the interrupt-driven nature of phones all that well. For example, the user might start typing in a note, then wind up taking a phone call. If our process is terminated, depending upon how the user tries getting back into our app, we might not have the note from our saved instance state.

A better approach, in many cases, is to save data in `onStop()`, when the activity moves into the background. If there is a chance that the user might not want the

partially-entered information, you could save it in a "side" area, such as a temporary file, and deal with it when the user returns to your app. Or, you could just update the real data store... which is what we will do here.

Edit the `onStop()` method in `NoteFragment` to look like the following:

```
@Override
public void onStop() {
  DatabaseHelper.getInstance(getActivity())
    .updateNote(getPosition(),
      editor.getText().toString());

  EventBus.getDefault().unregister(this);

  super.onStop();
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Here, we update the note. This is a bit inefficient, as we update the database even if the user did not change the text of the note, or even if the note is empty. That represents another optimization that a production-grade app might wish to pursue but is skipped here in the interests of simplicity.

If you build and run the app on a device or emulator, you will see the new "notes" toolbar button in the action bar:

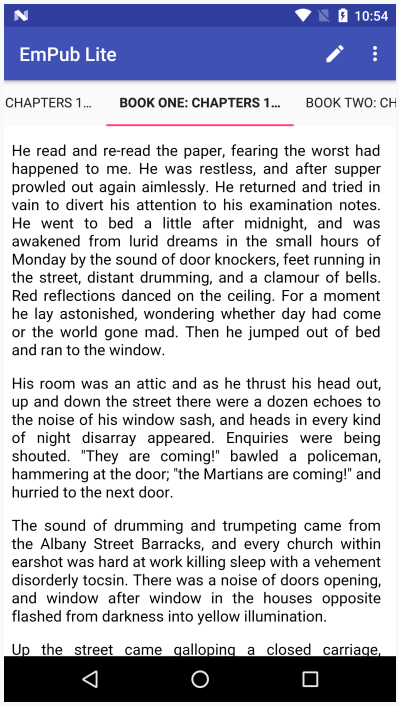


Figure 297: The New Action Bar Item

Tapping that will bring up the notes for whatever `ViewPager` position that you are on. Entering in some notes and pressing BACK to exit the activity will save those notes, which you will see again if you tap the action bar toolbar button again. If you change the notes, pressing BACK will save the changed notes in the database, to be viewed again later when you go back into the notes for that `ViewPager` position.

Step #13: Adding a Delete Action Bar Item

The only problem with this solution is that the notes never leave. While the user could manually delete everything in the `EditText`, it would be nice to make that perhaps a bit simpler. In this step, we will add an action bar item that will clear the `EditText` for the user.

Right-click over the `res/` directory and choose New > Vector Asset from the context menu. Click the Icon button and search for the “delete” icon:

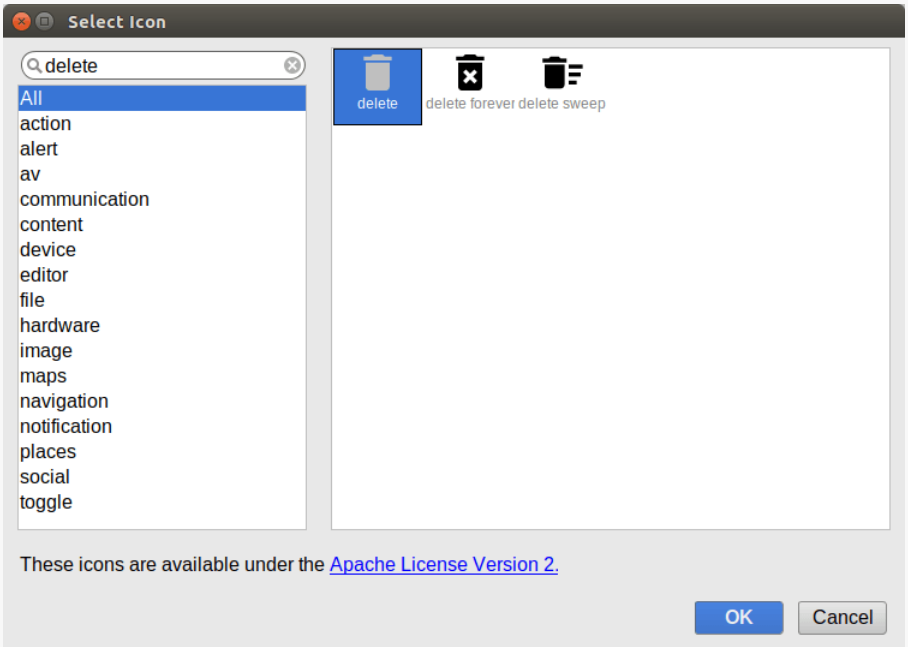


Figure 298: Asset Studio Icon Picker, with Delete Icon Selected

Click OK to close the icon picker. Change the name of the resource to `ic_delete_white_24dp`. Then click Next and Finish to save this drawable resource. Then, open `res/drawable/ic_delete_white_24dp.xml` and change the `android:fillColor` in the `<path>` element to be `#FFFFFFFF` instead of `#FF000000`:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
        android:width="24dp"
        android:height="24dp"
        android:viewportWidth="24.0"
        android:viewportHeight="24.0">
    <path
        android:fillColor="#FFFFFFFF"
        android:pathData="M6,19c0,1.1 0.9,2 2,2h8c1.1,0 2,-0.9
2,-2V7H6v12zM19,4h-3.5l-1,-1h-5l-1,1H5v2h14V4z"/>
</vector>
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/res/drawable/ic_delete_white_24dp.xml)

Next, add a new string resource, named `delete`, with a value like `Delete`.

Then, create a new resource, `res/menu/notes.xml`, to configure the action bar for the activity hosting our `NoteFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
```

```
    android:id="@+id/delete"
    android:icon="@drawable/ic_delete_white_24dp"
    android:showAsAction="ifRoom|withText"
    android:title="@string/delete">
  </item>
</menu>
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/res/menu/notes.xml)

This simply defines a single action bar item, with an ID of `delete`.

To do this, Android Studio users can right-click over the `res/menu/` directory and choose New > "Menu resource file" from the context menu. Fill in `notes.xml` in the "New Menu Resource File" dialog and click OK. Paste in the XML shown above into that file.

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

To let Android know that our `NoteFragment` wishes to participate in the action bar, we need to call `setHasOptionsMenu(true)` at some point. Add an `onCreate()` method to `NoteFragment` to handle this when the fragment is created:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setHasOptionsMenu(true);
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

That will trigger a call to `onCreateOptionsMenu()`, which we will need to add to `NoteFragment`:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.notes, menu);

  super.onCreateOptionsMenu(menu, inflater);
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

This just inflates our new resource for use in the options menu.

If the user taps on that toolbar button, `onOptionsItemSelected()` will be called, so we will need to add that as well to `NoteFragment`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId() == R.id.delete) {
    editor.setText(null);

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

Here, if the user tapped on our `delete` action bar item, we clear the `EditText` widget.

Step #14: Closing the NoteFragment When Deleted

However, tapping on that action bar item keeps the `NoteFragment` on the screen. It might be nice to automatically return to the book instead. However, the `NoteFragment` itself does not know how to do that, as something else (in this case, `NoteActivity`) put the `NoteFragment` on the screen. Hence, we need to pass the request to close the `NoteFragment` along to the proper party.

We could use another event object and our `EventBus`. In this case, we will demonstrate another approach: using the contract pattern to alert the hosting activity that the notes should be closed.

Define an *inner interface* in the `NoteFragment`, named `Contract`, as follows:

```
public interface Contract {
  void closeNotes();
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

You might put those lines immediately after the `public class NoteFragment...` line, before the declaration of any of the data members or methods, for example.

Then, add a private `getContract()` method, that casts the hosting `Activity` to the `Contract` interface:

```
  private Contract getContract() {
    return((Contract)getActivity());
  }
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

What we are doing here is enforcing that the activity that hosts our `NoteFragment` must implement the `NoteFragment.Contract` interface.

Then, add a call to `closeNotes()` on the `Contract` to our logic in `onOptionsItemSelected()`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
      editor.setText(null);
      getContract().closeNotes();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Now, when the user clicks on the `delete` action bar item, we clear the `EditText` and ask the hosting activity to get rid of us. Along the way, our `onPause()` will be called, causing us to clear the content of the `prose` column in our database row as well.

At this point, `NoteFragment` should resemble:

```
package com.commonsware.empublite;

import android.app.Fragment;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import org.greenrobot.eventbus.EventBus;
import org.greenrobot.eventbus.Subscribe;
```

```
import org.greenrobot.eventbus.ThreadMode;

public class NoteFragment extends Fragment {
  public interface Contract {
    void closeNotes();
  }

  private static final String KEY_POSITION="position";
  private EditText editor=null;

  static NoteFragment newInstance(int position) {
    NoteFragment frag=new NoteFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);

    return(frag);
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setHasOptionsMenu(true);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);

    editor=(EditText)result.findViewById(R.id.editor);

    return(result);
  }

  @Override
  public void onStart() {
    super.onStart();

    EventBus.getDefault().register(this);

    if (TextUtils.isEmpty(editor.getText())) {
      DatabaseHelper db=DatabaseHelper.getInstance(getActivity());
      db.loadNote(getPosition());
    }
```

```
  }

  @Override
  public void onStop() {
    DatabaseHelper.getInstance(getActivity())
      .updateNote(getPosition(),
        editor.getText().toString());

    EventBus.getDefault().unregister(this);

    super.onStop();
  }

  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.notes, menu);

    super.onCreateOptionsMenu(menu, inflater);
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
      editor.setText(null);
      getContract().closeNotes();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }

  @SuppressWarnings("unused")
  @Subscribe(threadMode =ThreadMode.MAIN)
  public void onNoteLoaded(NoteLoadedEvent event) {
    if (event.getPosition() == getPosition()) {
      editor.setText(event.getProse());
    }
  }

  private int getPosition() {
    return(getArguments().getInt(KEY_POSITION, -1));
  }

  private Contract getContract() {
    return((Contract)getActivity());
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

`NoteActivity` now must implement `NoteFragment.Contract` and implement `closeNotes()`. Modify `NoteActivity` to look like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;

public class NoteActivity extends Activity
  implements NoteFragment.Contract {
  public static final String EXTRA_POSITION="position";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      int position=getIntent().getIntExtra(EXTRA_POSITION, -1);

      if (position >= 0) {
        Fragment f=NoteFragment.newInstance(position);

        getFragmentManager().beginTransaction()
          .add(android.R.id.content, f).commit();
      }
    }
  }

  @Override
  public void closeNotes() {
    finish();
  }
}
```

(from EmPubLite-AndroidStudio/T14-Database/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteActivity.java)

This adds the `implements` keyword and the `closeNotes()` implementation, which just finishes the `NoteActivity`, returning control to the `EmPubLiteActivity`.

If you run this in a device or emulator, and you go into the notes, you will see our delete toolbar button:

Figure 299: The New Action Bar Item

Tapping that toolbar button will clear the note and close the activity, returning you to the book.

In Our Next Episode…

… we will allow the user to share a chapter's notes with somebody else.

Internet Access

The expectation is that most, if not all, Android devices will have built-in Internet access. That could be WiFi, cellular data services (EDGE, 3G, etc.), or possibly something else entirely. Regardless, most people — or at least those with a data plan or WiFi access — will be able to get to the Internet from their Android phone.

Not surprisingly, the Android platform gives developers a wide range of ways to make use of this Internet access. Some offer high-level access, such as the integrated WebKit browser component (`WebView`) we saw in an earlier chapter. If you want, you can drop all the way down to using raw sockets. Or, in between, you can leverage APIs — both on-device and from 3rd-party JARs — that give you access to specific protocols: HTTP, XMPP, SMTP, and so on.

The emphasis of this book is on the higher-level forms of access: the WebKit component and Internet-access APIs, as busy coders should be trying to reuse existing components versus rolling one's own on-the-wire protocol wherever possible.

DIY HTTP

In many cases, your only viable option for accessing some Web service or other HTTP-based resource is to do the request yourself. The Google-endorsed API for doing this nowadays in Android is to use the classic `java.net` classes for HTTP operation, centered around `HttpUrlConnection`. There is quite a bit of material on this already published, as these classes have been in Java for a long time. The focus here is in showing how this works in an Android context.

Note, however, that you may find it easier to use some HTTP client libraries that handle various aspects of the Internet access for you, as will be described later in this chapter.

A Sample Usage of HttpUrlConnection

This chapter walks through several implementations of a Stack Overflow client application. The app has a single activity, with a single `ListFragment`. The app will load the latest block of Stack Overflow questions tagged with `android`, using the Stack Exchange public API. Those questions will be shown in the list, and tapping on a question will bring up the Web page for that question in the user's default Web browser.

All implementations of the app have the same core UI logic. What differs is in how each handles the Internet access. In this section, we will take a look at the `Internet/HURL` sample project, which uses `HttpUrlConnection` to retrieve the questions from the Stack Exchange Web service API.

Asking Permission

To do anything with the Internet (or a local network) from your app, you need to hold the `INTERNET` permission. This includes cases where you use things like `WebView` — if your *process* needs network access, you need the `INTERNET` permission.

Hence, the manifest for our sample project contains the requisite `<uses-permission>` declaration:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Creating Your Data Model

The Stack Exchange Web service API returns JSON in response to various queries. Hence, we need to create Java classes that mirror that JSON structure. In particular, many of the examples will be using Google's Gson to populate those data models automatically based upon its parsing of the JSON that we receive from the Web service.

In our case, we are going to use a specific endpoint of the Stack Exchange API, referred to as `/questions` after the distinguishing portion of the path. The

documentation for this endpoint can be found in the Stack Exchange API documentation.

We will examine the URL for the endpoint a bit later in this section.

The results we get for issuing a `GET` request for the URL is a JSON structure (here showing a single question, to keep the listing short):

```
{
  "items": [
    {
      "question_id": 17196927,
      "creation_date": 1371660594,
      "last_activity_date": 1371660594,
      "score": 0,
      "answer_count": 0,
      "title": "ksoap2 failing when in 3G",
      "tags": [
        "android",
        "ksoap2",
        "3g"
      ],
      "view_count": 2,
      "owner": {
        "user_id": 773259,
        "display_name": "SparK",
        "reputation": 513,
        "user_type": "registered",
        "profile_image": "http://www.gravatar.com/avatar/
511b37f7c313984e624dd76e8cb9faa6?d=identicon&r=PG",
        "link": "http://stackoverflow.com/users/773259/spark"
      },
      "link": "http://stackoverflow.com/questions/17196927/ksoap2-failing-when-in-3g",
      "is_answered": false
    }
  ],
  "quota_remaining": 9991,
  "quota_max": 10000,
  "has_more": true
}
```

NOTE: Some of the longer URLs will word-wrap in the book, but they are on a single line in the actual JSON. Honest.

We get back a JSON object, where our questions are found under the name of `items`. `items` is a JSON array of JSON objects, where each JSON object represents a single

question, with fields like `title` and `link`. The question JSON object has an embedded `owner` JSON object with additional information.

We do not necessarily need all of this information. In fact, for this first version of the sample, all we really need are the `title` and `link` of each entry in the `items` array.

The key is that, by default, the data members in our Java data model must *exactly* match the JSON keys for the JSON objects.

So, we have an `Item` class, representing the information from an individual entry in the `items` array:

```
package com.commonsware.android.hurl;

public class Item {
  String title;
  String link;

  @Override
  public String toString() {
    return(title);
  }
}
```

(from Internet/HURL/app/src/main/java/com/commonsware/android/hurl/Item.java)

However, our Web service does not return the `items` array directly. `items` is the key in a JSON object that is the actual JSON returned by Stack Exchange. So, we need another Java class that contains the data members we need from that outer JSON object, here named `SOQuestions` (for lack of a better idea for a name...):

```
package com.commonsware.android.hurl;

import java.util.List;

public class SOQuestions {
  List<Item> items;
}
```

(from Internet/HURL/app/src/main/java/com/commonsware/android/hurl/SOQuestions.java)

Having an `items` data member that is a `List` of `Item` tells GSON that we are expecting the JSON object to be used for `SOQuestions` to have a JSON array, named `items`, where each element in that array should get mapped to `Item` objects.

A Thread for Loading

We need to do the network I/O on a background thread, so we do not tie up the main application thread. To that end, the sample app has a `LoadThread` that loads our questions:

```
package com.commonsware.android.hurl;

import android.util.Log;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import com.google.gson.Gson;
import de.greenrobot.event.EventBus;

class LoadThread extends Thread {
  static final String SO_URL=
      "https://api.stackexchange.com/2.1/questions?"
          + "order=desc&sort=creation&site=stackoverflow&tagged=android";

  @Override
  public void run() {
    try {
      HttpURLConnection c=
          (HttpURLConnection)new URL(SO_URL).openConnection();

      try {
        InputStream in=c.getInputStream();
        BufferedReader reader=
            new BufferedReader(new InputStreamReader(in));
        SOQuestions questions=
            new Gson().fromJson(reader, SOQuestions.class);

        reader.close();

        EventBus.getDefault().post(new QuestionsLoadedEvent(questions));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
      finally {
        c.disconnect();
      }
    }
```

```
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
}
```

(from Internet/HURL/app/src/main/java/com/commonsware/android/hurl/LoadThread.java)

`LoadThread`:

- Creates an `HttpUrlConnection` by creating a `URL` for our Stack Exchange API endpoint and opening a connection
- Creates a `BufferedReader` wrapped around the `InputStream` from the HTTP connection
- Parses the JSON we get back from that HTTP request via a `Gson` instance, loading the data into an instance of our `SOQuestions`
- Close the `BufferedReader` (and the `InputStream` by extension)
- Post a `QuestionsLoadedEvent` to greenrobot's `EventBus`, to let somebody know that our questions exist
- Log messages to Logcat in case of errors

`QuestionsLoadedEvent` is a simple wrapper around an `SOQuestions` instance, serving as an event class for use with `EventBus`:

```
package com.commonsware.android.hurl;

public class QuestionsLoadedEvent {
  final SOQuestions questions;

  QuestionsLoadedEvent(SOQuestions questions) {
    this.questions=questions;
  }
}
```

(from Internet/HURL/app/src/main/java/com/commonsware/android/hurl/QuestionsLoadedEvent.java)

A Fragment for Questions

The sample app has a `QuestionsFragment` that should display these loaded questions:

```
package com.commonsware.android.hurl;

import android.app.ListFragment;
```

```
import android.os.Bundle;
import android.text.Html;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import java.util.List;
import de.greenrobot.event.EventBus;

public class QuestionsFragment extends ListFragment {
  public interface Contract {
    void onQuestion(Item question);
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
    new LoadThread().start();
  }

  @Override
  public void onResume() {
    super.onResume();
    EventBus.getDefault().register(this);
  }

  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);
    super.onPause();
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    Item item=((ItemsAdapter)getListAdapter()).getItem(position);

    ((Contract)getActivity()).onQuestion(item);
  }

  public void onEventMainThread(QuestionsLoadedEvent event) {
    setListAdapter(new ItemsAdapter(event.questions.items));
  }

  class ItemsAdapter extends ArrayAdapter<Item> {
    ItemsAdapter(List<Item> items) {
```

```
      super(getActivity(), android.R.layout.simple_list_item_1, items);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      TextView title=(TextView)row.findViewById(android.R.id.text1);

      title.setText(Html.fromHtml(getItem(position).title));

      return(row);
    }
  }
}
```

(from Internet/HURL/app/src/main/java/com/commonsware/android/hurl/QuestionsFragment.java)

In `onCreate()`, we mark that this fragment should be retained and fork the `LoadThread`. Hence, once we have our questions, our retained fragment will hold onto that model data for us, and we avoid duplicating the `LoadThread` if a configuration change occurs sometime after our fragment was initially created.

In `onResume()` and `onPause()`, we register and unregister from the `EventBus`. Our `onEventMainThread()` method will be called when the `QuestionsLoadedEvent` is raised by `LoadThread`, and there we hold onto the loaded questions and populate the `ListView`. We use an `ItemsAdapter`, which knows how to render an `Item` as a simple `ListView` row showing the question title. The `ItemsAdapter` uses `Html.fromHtml()` to populate the `ListView` rows, not because Stack Overflow hands back titles with HTML tags, but because Stack Overflow hands back titles with HTML *entity references*, and `Html.fromHtml()` should handle many of those.

And, in `onListItemClick()`, we find the `Item` associated with the row that the user clicked upon, then call an `onQuestion()` method on our hosting activity. That activity needs to implement the `Contract` interface, so we can call the `onQuestion()` method on whatever activity happens to host this fragment.

An Activity for Orchestration

`MainActivity` sets up the fragment in `onCreate()` and handles the click events in `onQuestion()`:

```
package com.commonsware.android.hurl;

import android.app.Activity;
```

```
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import de.greenrobot.event.EventBus;

public class MainActivity extends Activity
  implements QuestionsFragment.Contract {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new QuestionsFragment()).commit();
    }
  }

  @Override
  public void onQuestion(Item question) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(question.link)));
  }
}
```

(from Internet/HURL/app/src/main/java/com/commonsware/android/hurl/MainActivity.java)

Hence, `MainActivity` is serving in an orchestration role. `QuestionsFragment` is a local controller, handling direct events raised by its widgets (a `ListView`). `MainActivity` is responsible for handling events that transcend an individual fragment — in this case, it starts a browser to view the clicked-upon question.

The result is a simple `ListView` showing questions:

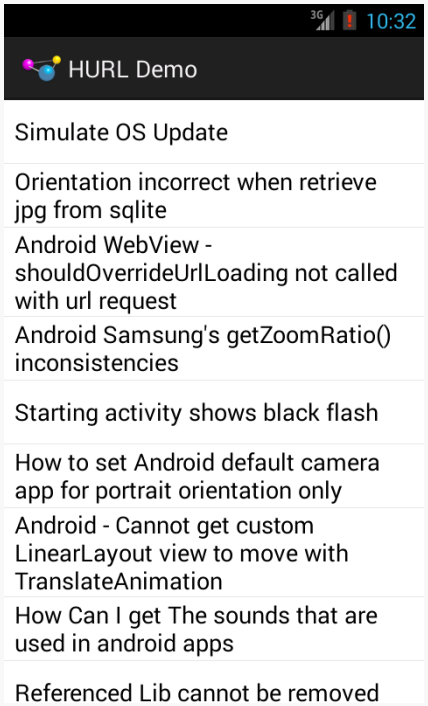


Figure 300: HURLDemo, Showing Stack Overflow Questions

What Android Brings to the Table

Google has augmented `HttpUrlConnection` to do more stuff to help developers. Notably:

- It automatically uses GZip compression on requests, adding the appropriate HTTP header and automatically decompressing any compressed responses (added in Android 2.3)
- It uses Server Name Indication to help work with several HTTPS hosts sharing a single IP address
- API Level 13 (Android 4.0) added an `HttpResponseCache` implementation of the `java.net.ResponseCache` base class, that can be installed to offer transparent caching of your HTTP requests.

Also, courtesy of some third-party code (OkHttp) that we will discuss shortly, `HttpUrlConnection` also supports the SPDY protocol and HTTP/2 for accelerating Web content distribution over SSL as of Android 4.4.

Testing with StrictMode

`StrictMode`, mentioned in the chapter on files, can also report on performing network I/O on the main application thread. More importantly, on Android 3.0 and higher, by default, Android will crash your app with a `NetworkOnMainThreadException` if you try to perform network I/O on the main application thread.

Hence, it is generally a good idea to test your app, either using `StrictMode` yourself or using a suitable emulator, to make sure that you are not performing network I/O on the main application thread.

What About HttpClient?

Android also contains — or used to contain – a mostly-complete copy of version 4.0.2beta of the Apache HttpClient library. Many developers use this, as they prefer the richer API offered by this library over the somewhat more clunky approach used by `java.net`. And, truth be told, this was the more stable option prior to Android 2.3.

There are a few reasons why this is no longer recommended, for Android 2.3 and beyond:

- The core Android team is better able to add capabilities to the `java.net` implementation while maintaining backwards compatibility, because its API is more narrow.
- The problems previously experienced on Android with the `java.net` implementation have largely been fixed.
- The Apache HttpClient project continuously evolves its API. This means that Android will continue to fall further and further behind the latest-and-greatest from Apache, as Android insists on maintaining the best possible backwards compatibility and therefore cannot take on newer-but-different HttpClient versions.
- Google officially deprecated this API in Android 5.1.
- Google officially removed this API in Android 6.0.

If you have legacy code that uses the HttpClient API, please consider using Apache's standalone edition of HttpClient for Android.

And, if you cannot do any of that, and you are using Gradle for your builds (e.g., you are using Android Studio's default settings), you can add `useLibrary 'org.apache.http.legacy'` to the `android` closure to give you access to Android's stock HttpClient API:

```
android {
    useLibrary 'org.apache.http.legacy'
    // other settings go here
}
```

However, usually, using a standalone edition should be reasonably practical.

For example, the `Internet/HttpClient` sample project is a clone of the `HttpURLConnection` sample from earlier in this chapter, revised to use HttpClient. More specifically, it uses the `cz.msebera.android` packaging of Apache HttpClient for Android:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'de.greenrobot:eventbus:2.2.1'
    implementation 'com.google.code.gson:gson:2.8.0'
    implementation 'cz.msebera.android:httpclient:4.4.1.1'
}

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 16
        targetSdkVersion 25
    }
}
```

(from Internet/HttpClient/app/build.gradle)

The classes are all the same as in the equivalent Apache HttpClient code, except that `org.apache` is replaced by `cz.msebera.android`.

The code that you might have used with Android's built-in HttpClient will not directly work on newer versions of Apache's HttpClient, due to API differences. However, it is fairly close in most common places, requiring slight modifications, usually to improve the API.

For example, here is the `LoadThread` class from before, revised to use HttpClient:

```
package com.commonsware.android.httpclient;

import android.util.Log;
import com.google.gson.Gson;
import java.io.IOException;
import cz.msebera.android.httpclient.client.HttpClient;
import cz.msebera.android.httpclient.client.methods.HttpGet;
import cz.msebera.android.httpclient.impl.client.BasicResponseHandler;
import cz.msebera.android.httpclient.impl.client.HttpClientBuilder;
import cz.msebera.android.httpclient.impl.conn.PoolingHttpClientConnectionManager;
import de.greenrobot.event.EventBus;

class LoadThread extends Thread {
  static final String SO_URL=
      "https://api.stackexchange.com/2.1/questions?"
          + "order=desc&sort=creation&site=stackoverflow&tagged=android";

  @Override
  public void run() {
    try {
      HttpClient client=HttpClientBuilder.create()
        .setConnectionManager(
          new PoolingHttpClientConnectionManager())
        .build();
      HttpGet get=new HttpGet(SO_URL);

      try {
        String result=client.execute(get, new BasicResponseHandler());
        SOQuestions questions=
            new Gson().fromJson(result, SOQuestions.class);

        EventBus.getDefault().post(new QuestionsLoadedEvent(questions));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
}
```

(from Internet/HttpClient/app/src/main/java/com/commonsware/android/httpclient/LoadThread.java)

We start by getting an `HttpClient` instance from an `HttpClientBuilder`. Particularly if you use the `PoolingHttpClientConnectionManager`, as shown here, the `HttpClient` instance is designed to be shared among multiple threads, and so could be used as a singleton. Here, we only use it once, so it is merely a local variable in the `run()` method.

From there, we:

- Create an `HttpGet` object to represent the Stack Exchange API `GET` request
- Tell the `HttpClient` to `execute()` the `HttpGet`, passing the response through a `BasicResponseHandler`, which returns a `String` containing our JSON (assuming there is no server error)
- Pass that `String` through Gson to get our `SOQuestions`, much as before

HTTP via DownloadManager

If your objective is to download some large file, you may be better served by using the `DownloadManager` added to Android 2.3, as it handles a lot of low-level complexities for you. For example, if you start a download on WiFi, and the user leaves the building and the device fails over to some form of mobile data, you need to reconnect to the server and either start the download again or use some content negotiation to pick up from where you left off. `DownloadManager` handles that.

However, `DownloadManager` is dependent upon some broadcast `Intent` objects, a technique we have not discussed yet, so we will delay covering `DownloadManager` until later in the book.

Using Third-Party JARs

To some extent, the best answer is to not write the code yourself, but rather use some existing JAR that handles both the Internet I/O and any required data parsing. This is commonplace when accessing public Web services — either because the firm behind the Web service has released a JAR, or because somebody in the community has released a JAR for that Web service.

Examples include:

- Using JTwitter to access Twitter's API
- Using Amazon's JAR to access various AWS APIs, including S3, SimpleDB, and SQS
- Using the Dropbox SDK for accessing DropBox folders and files

However, beyond the classic potential JAR problems, you may encounter another when it comes to using JARs for accessing Internet services: versioning. For example:

- JTwitter bundles the `org.json` classes in its JAR, which will be superseded by Android's own copy, and if the JTwitter version of the classes have a different API, JTwitter could crash.

- Libraries dependent upon HttpClient might be dependent upon a version with a different API (e.g., 4.1.1) than is in Android (4.0.2 beta).

Try to find JARs that have been tested on Android and are clearly supported as such by their author. Lacking that, try to find JARs that are open source, so you can tweak their implementation if needed to add Android support.

Later in this chapter, we will review another class of third-party JARs, ones that are more general-purpose than things like JTwitter, but still offer to simplify HTTP processing.

SSL

Of course, if you are thinking about HTTP, you really should be thinking about HTTPS — SSL-encrypted HTTP operations.

Normally, SSL "just works", by using an `https://` URL. Hence, typically, there is little that you need to do to enable simple encryption.

However, there are other aspects of SSL to consider, including:

- What if the server is not using an SSL certificate that Android will honor, such as a self-signed certificate?
- What about man-in-the-middle attacks, hacked certificate authorities, and the like?

The trails contain a chapter dedicated to SSL that you are encouraged to read, so that this chapter does not get crazy-long.

Using HTTP Client Libraries

Often times, writing Internet access code is a pain in various body parts.

Not surprisingly, there are a variety of third-party libraries designed to assist with this. Some are designed to provide access to a specific API, such as the ones mentioned earlier in this chapter. However, others are more general-purpose, designed to make writing HTTP operations a bit easier, by handling things like:

- Retries (e.g., device failed over from WiFi to mobile data mid-transaction)

- Threading (e.g., handling doing the Internet work on a background thread for you)
- Data parsing and marshaling, for well-known formats (e.g., JSON)

In this section, we will look at three libraries that exemplify this approach: OkHttp, Retrofit, and Picasso. Later, we will see other libraries that you might wish to investigate, including Google's own Volley HTTP client API.

OkHttp

OkHttp uses a modified clone of the standard `HttpUrlConnection` to offer many performance improvements. Most notable is its support for SPDY, a Google sponsored enhanced version of HTTP, going beyond classic HTTP "keep-alive" support to allow for many requests and responses to be delivered over the same socket connection. This, in turn, evolved into HTTP/2. Many Google APIs are served by SPDY- or HTTP/2-capable servers, and HTTP/2 is gaining popularity overall.

Beyond that, OkHttp wraps up common HTTP performance-improvement patterns, such as GZIP compression, response caching, and connection pooling. It also is more aware of "real world" connection issues, like mis-configured proxy servers and the like.

Note that a version of OkHttp lies behind the standard implementation of `HttpUrlConnection` in Android 4.4 and higher — this is where Android's SPDY support comes from.

OkHttp has changed over the years. OkHttp came to prominence in its 1.x and 2.x versions, with a reasonably stable API. OkHttp 3.x is the current generation, with a richer but slightly different API.

OkHttp 2.x

The `Internet/OkHttp` sample project is a clone of the Stack Overflow sample shown earlier in this chapter. The original sample used `HttpURLConnection` to download the Stack Exchange Web service data. This revised sample replaces that with OkHttp.

First, we need to add a dependency on OkHttp to our `app/` module's `build.gradle` file:

```
dependencies {
    implementation 'de.greenrobot:eventbus:2.4.0'
    implementation 'com.google.code.gson:gson:2.8.0'
    implementation 'com.squareup.okhttp:okhttp:2.4.0'
}
```

(from Internet/OkHttp/app/build.gradle)

OkHttp offers two basic flavors of HTTP API: synchronous and asynchronous. With a synchronous call, the call blocks until the HTTP I/O is completed (or, at least, the headers are downloaded). With an asynchronous call, that initial pulse of network I/O is handled on a background thread. The general rule of thumb is:

- If you can work with the raw HTTP response, and it's short, use the asynchronous API, as it saves you from having to fuss with your own thread
- If the response may be long or requires significant post-retrieval work (e.g., parsing), use your own background thread and use the synchronous API

In our case, we need to parse the JSON using Gson, and so the second approach is the better answer. This has the side benefit of limiting our Java changes to only be in `LoadThread`:

```
package com.commonsware.android.okhttp;

import android.util.Log;
import com.google.gson.Gson;
import com.squareup.okhttp.OkHttpClient;
import com.squareup.okhttp.Request;
import com.squareup.okhttp.Response;
import java.io.BufferedReader;
import java.io.Reader;
import de.greenrobot.event.EventBus;

class LoadThread extends Thread {
  static final String SO_URL=
      "https://api.stackexchange.com/2.1/questions?"
          + "order=desc&sort=creation&site=stackoverflow&tagged=android";

  @Override
  public void run() {
    try {
      OkHttpClient client=new OkHttpClient();
      Request request=new Request.Builder().url(SO_URL).build();
      Response response=client.newCall(request).execute();

      if (response.isSuccessful()) {
```

```
        Reader in=response.body().charStream();
        BufferedReader reader=new BufferedReader(in);
        SOQuestions questions=
            new Gson().fromJson(reader, SOQuestions.class);

        reader.close();

        EventBus.getDefault().post(new QuestionsLoadedEvent(questions));
      }
      else {
        Log.e(getClass().getSimpleName(), response.toString());
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
}
```

(from Internet/OkHttp/app/src/main/java/com/commonsware/android/okhttp/LoadThread.java)

Our revised `run()` method creates an instance of `OkHttpClient`. This is your gateway for performing HTTP requests.

An individual request, not surprisingly, is represented by a `Request` object, usually created using a `Request.Builder`. The `url()` method on the `Builder` is where you supply the URL to be retrieved via a `GET` request. There are other methods on `Builder`, such as `post()`, which supplies additional data for the request and converts it into a `POST` request.

To actually perform the synchronous request, call `newCall().execute()` on the `OkHttpClient`, passing in the `Request` to the `newCall()` method. This gives you a `Response` object, which should return `true` from `isSuccessful()`. `false` would indicate some sort of a problem, such as an HTTP 404 response code.

Given a successful `Response`, you can get at the body of the response via a `body()` method. This returns a `ResponseBody`, which offers three main ways to get at the response body itself:

- `string()` returns the entire response as a `String`, which is only really suitable for short text responses
- `byteStream()` returns an `InputStream` on the raw bytes of the response
- `charStream()`, despite its name, returns a `Reader` on the characters of the response, taking into account the response's character encoding (e.g., UTF-8)

Here, we use `charStream()` to get a `Reader`, which we then wrap in a `BufferedReader`. The rest of the `run()` method is pretty much the same as the original, asking Gson to parse the response and posting a `QuestionsLoadedEvent` to get the questions over to our fragment on the main application thread.

OkHttp3

For simple projects like this one, there is no substantial change required for OkHttp3. The `Internet/OkHttp3` sample project is a clone of the `Internet/OkHttp` project, adjusted to work with OkHttp3. There are only two such adjustments:

- Update `build.gradle` to point to the `com.squareup.okhttp3` artifact:

```
dependencies {
    implementation 'de.greenrobot:eventbus:2.4.0'
    implementation 'com.google.code.gson:gson:2.8.2'
    implementation 'com.squareup.okhttp3:okhttp:3.9.0'
}
```

(from Internet/OkHttp3/app/build.gradle)

- Update the `import` statements to use `okhttp3` as a package, instead of `com.squareup.okhttp`

Because the artifact and imports are both different, it is technically possible to have *both* OkHttp 2.x and 3.x in a project at one time. That is not a great long-term solution, but it may prove useful, particularly if you are using other libraries that happen to depend on different generations of OkHttp.

OkHttp 1.x and 2.x supported compatibility layers that modeled `HttpURLConnection` and Apache's `HttpClient`. These have been discontinued with OkHttp3.

Retrofit

Many times, when working with HTTP requests, our needs are fairly simple: just retrieve some JSON (or other structured data, such as XML) from some Web service, or perhaps upload some JSON to that Web service.

Retrofit is designed to simplify this, by handling the data parsing and marshaling for us, along with the HTTP operations and (optionally) background threading. We are left with a fairly natural-looking Java API to send/receive Java objects to/from the

Web service. Retrofit accomplishes this through the cunning use of annotations, reflection, and, where available, OkHttp itself.

There are two major generations of the Retrofit API: 1.x and 2.x. Both share the same objectives and general capabilities. Retrofit 1.x was fairly popular, and so you will see a fair bit of code still using that, even though new development should strongly consider the newer 2.x generation. In this section, we will review both, looking first at 1.x and then seeing what changed in 2.x.

Retrofit 1.x

To demonstrate Retrofit 1.x (and set the stage for Retrofit 2.x), let's review the `HTTP/Retrofit` sample project. This project is a clone of the preceding one, this time using Retrofit for retrieving and parsing our Stack Overflow questions.

Note that Stack Overflow happens to use JSON as its data format, which works nicely with Retrofit, as JSON is its default data format. However, you can supply your own conversion logic, to convert data to/from other formats, such as XML or Protocol Buffers.

Dependencies

The `com.squareup.retrofit:retrofit` artifact is used for Retrofit 1.x. The last published release version of Retrofit 1.x was 1.9.0, and this project includes that dependency:

```
dependencies {
    implementation 'com.squareup.retrofit:retrofit:1.9.0'
}
```

(from HTTP/Retrofit/app/build.gradle)

By default, Retrofit uses Google's Gson for its JSON parsing. The `com.squareup.retrofit:retrofit` artifact has Gson as a dependency, so requiring `com.squareup.retrofit:retrofit` will also pull in a compatible version of Gson.

Creating Your Service Interface

The first thing we need is to tell Retrofit more about where our JSON is coming from. To do this, we need to create a Java interface with some specific Retrofit-supplied annotations, documenting:

- the HTTP operations that we wish to perform
- the path (and, if needed, query parameters) to apply an HTTP operation to
- the per-request data to configure the HTTP operation, such as the dynamic portions of the path for a REST-style API, or additional query parameters to attach to the URL
- what object should be used for pouring the HTTP response into

For example, let's take a look at `StackOverflowInterface`, our interface for making a query of Stack Exchange's API to get questions from Stack Overflow:

```
package com.commonsware.android.retrofit;

import retrofit.Callback;
import retrofit.http.GET;
import retrofit.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  void questions(@Query("tagged") String tags, Callback<SOQuestions> cb);
}
```

(from HTTP/Retrofit/app/src/main/java/com/commonsware/android/retrofit/StackOverflowInterface.java)

Each method in the interface should have an annotation identifying the HTTP operation to perform, such as `@GET` or `@POST`. The parameter to the annotation is the path for the request and any fixed query parameters. In our case, we are using the path documented by Stack Exchange for retrieving questions (`/2.1/questions`), plus some fixed query parameters:

- `order` for whether the results should be ascending (`asc`) or descending (`desc`)
- `sort` to indicate how the questions should be sorted, such as `creation` to sort by time when the question was posted
- `site` to indicate what Stack Exchange site we are querying (e.g., `stackoverflow`)

The method name can be whatever you want.

If you have additional query parameters that vary dynamically, you can use the `@Query` annotation on `String` parameters to have them be added to the end of the URL. In our case, the `tagged` query parameter will be added with whatever the `tags` parameter is to our `questions()` method.

Similarly, you can use `{name}` placeholders for path segments, and replace those at runtime via `@Path`-annotated parameters to the method.

To get results back, and indicate the data type for those results, you have two choices:

1. Have the method return the data type you wish, in which case when we eventually call this method, the HTTP operation will be performed synchronously, blocking our method call
2. Pass a `Callback` parameter, declared with the desired data type (e.g., `SOQuestions`), in which case when we eventually call this method, the HTTP operation will be performed on a background thread, with the results delivered to us asynchronously via a custom `Callback` implementation that we will supply

In this case, we are electing to let Retrofit handle the threading for us, so we supply a `Callback` and have the method return `void`.

Curiously, we will never create an implementation of the `StackOverflowInterface` ourselves. Instead, Retrofit generates one for us, with code that implements our requested behaviors.

Creating the RestAdapter

To use this generated `StackOverflowInterface`, and to actually perform these operations, we need to create an instance of a `RestAdapter`. Usually, you will do this via a `RestAdapter.Builder`, to configure what you want done.

The biggest thing you will provide to `RestAdapter.Builder` is the server tied to these HTTP operations. Calling `setEndpoint()` allows you to specify the scheme, host, and port to be attached to the rest of the URL, coming from your interface. For example, we need to make our requests of the `https://api.stackexchange.com` server, so we have:

```
    RestAdapter restAdapter=
        new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com")
                                 .build();
```

(from HTTP/Retrofit/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

Other methods on `RestAdapter.Builder` include:

- `setConverter()`, if your payloads are not in JSON format, but something else
- `setExecutors()`, to provide `Executor` objects (e.g., instances of `ThreadPoolExecutor`) to be used for requests and callbacks
- `setLog()` and `setDebug()` for controlling log output

When you are done configuring the `RestAdapter.Builder`, call `build()` to get the resulting `RestAdapter`.

Making Requests

Given a configured `RestAdapter`, you can retrieve an implementation of your API interface by calling the `create()` method:

```
    StackOverflowInterface so=
        restAdapter.create(StackOverflowInterface.class);
```

(from HTTP/Retrofit/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

You can then use the resulting interface-typed object no differently than you would any other Java object, despite the fact that you never wrote an implementation of that interface yourself.

In our case, we can call the `questions()` method, supplying the tag (or tags) from which we wish to receive recent questions:

```
    so.questions("android", this);
```

(from HTTP/Retrofit/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

The second parameter to `questions()` is an implementation of `Callback`, to receive asynchronous results from our HTTP `GET` request. `Callback` requires two methods, `success()` and `failure()`. `success()` takes two parameters: the data type you indicated in the interface (e.g., `SOQuestions`) representing the parsed results of the HTTP request, and a `Response` object containing other information from the HTTP response, such as headers:

```
@Override
public void success(SOQuestions questions, Response response) {
  setListAdapter(new ItemsAdapter(questions.items));
}
```

(from HTTP/Retrofit/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

Here, we update our `ListView` with an `ItemsAdapter` based upon the received questions.

`failure()` takes a single parameter, an instance of `RetrofitError`, which is an `Exception` providing details of something that went wrong in the HTTP request (e.g., authorization was denied). You can handle that no differently than you might other exceptions from elsewhere in your app, to let the user know that something went wrong. In this case, we take the crude-but-easy approach of showing a `Toast` and logging the details to Logcat:

```
  @Override
  public void failure(RetrofitError exception) {
    Toast.makeText(getActivity(), exception.getMessage(),
                   Toast.LENGTH_LONG).show();
    Log.e(getClass().getSimpleName(),
          "Exception from Retrofit request to StackOverflow", exception);
  }
```

(from HTTP/Retrofit/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

Retrofit 2.x

The `HTTP/Retrofit2` sample project is a clone of the preceding one, updated to use Retrofit 2.x (along with 3.x of greenrobot's EventBus).

There were a number of changes made in Retrofit 2.x, starting with the dependency. With Retrofit 1.x, Retrofit would automatically pull in a compatible version of Gson, and it would do so all the time (unless you took specific steps in Gradle to block that). In Retrofit 2.x, the parsing logic is now handled by plugins, as Retrofit 2.x has direct support for a wide range of parsers, for JSON (e.g., Gson, Jackson), XML (e.g., SimpleXML), and Protocol Buffers. If you request the artifact for one of those plugins, that artifact will pull in a compatible version of both Retrofit 2.x and the associated "converter" code (e.g., Gson). To allow apps to support both Retrofit 1.x and 2.x in the same project, Retrofit 2.x moved to the `com.squareup.retrofit2` group ID, with various artifacts in there. So, this revised sample app pulls in the `converter-gson` artifact, to get Retrofit with Gson support:

```
dependencies {
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
}
```

(from HTTP/Retrofit2/app/build.gradle)

Also, to allow projects to have both Retrofit 1.x and 2.x, all of the Retrofit 2.x classes are in a `retrofit2` Java package, distinct from the `retrofit` package used for Retrofit 1.x.

With Retrofit 1.x, our service interface would use two different method signatures for synchronous and asynchronous operation. Synchronous methods would return the desired object type (e.g., `SOQuestions`), while the asynchronous methods would return `void` and take a `Callback` as a parameter. Retrofit 2.x simplifies this, having all service interface method signatures return a `Call` object of the desired type:

```
package com.commonsware.android.retrofit;

import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.http.GET;
import retrofit2.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  Call<SOQuestions> questions(@Query("tagged") String tags);
}
```

(from HTTP/Retrofit2/app/src/main/java/com/commonsware/android/retrofit/StackOverflowInterface.java)

For synchronous work, you `execute()` the `Call`; for asynchronous work, you `enqueue()` the `Call`, providing a `Callback` object as a parameter to `enqueue()`.

Retrofit 1.x had a `RestAdapter` that you would use to configure the general Retrofit behavior, then use `create()` to get implementations of the service interface. Retrofit 2.x uses a similar system, but it is now a `Retrofit` object, created via a `Retrofit.Builder`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
        super.onCreateView(inflater, container, savedInstanceState);

    setRetainInstance(true);

    Retrofit retrofit=
      new Retrofit.Builder()
        .baseUrl("https://api.stackexchange.com")
        .addConverterFactory(GsonConverterFactory.create())
```

```
      .build();
  StackOverflowInterface so=
    retrofit.create(StackOverflowInterface.class);

  so.questions("android").enqueue(this);

  return(result);
}
```

(from HTTP/Retrofit2/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

Note the call to `addConverterFactory()` on the `Retrofit.Builder`. Since Retrofit 2.x no longer assumes that you want to use any particular parser, you have to specifically supply an instance of a `Converter`. `GsonConverterFactory` is in the `converter-gson` artifact and gives us a `GsonConverter` that we can create via `create()` on the factory.

The `Callback` interface is also slightly different, with `onResponse()` and `onFailure()` methods replacing the Retrofit 1.x `success()` and `failure()` methods:

```
@Override
public void onResponse(Call<SOQuestions> call,
                       Response<SOQuestions> response) {
  setListAdapter(new ItemsAdapter(response.body().items));
}

@Override
public void onFailure(Call<SOQuestions> call, Throwable t) {
  Toast.makeText(getActivity(), t.getMessage(),
    Toast.LENGTH_LONG).show();
  Log.e(getClass().getSimpleName(),
    "Exception from Retrofit request to StackOverflow", t);
}
```

(from HTTP/Retrofit2/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

In `onResponse()`, to get the results of the Retrofit request, call `body()` on the supplied `Response` object.

So, overall, Retrofit 2.x does the same sort of work as does Retrofit 1.x, just with a somewhat altered set of artifacts and APIs.

Picasso

Sometimes, what you want to download is not JSON, or XML, or any sort of structured data.

Sometimes, it is an image.

For example, Stack Overflow users have avatars. In our sample app, it might be nice to display the avatar of the user who asked the question.

Picasso is a library from Square that is designed to help with asynchronously loading images, whether those images come from HTTP requests, local files, a `ContentProvider`, etc. In addition to doing the loading asynchronously, Picasso simplifies many operations on those images, such as:

- Caching the results in memory (or optionally on disk for HTTP requests)
- Displaying placeholder images while the real images are being loaded, and displaying error images if there was a problem in loading the image (e.g., invalid URL)
- Transforming the image, such as resizing or cropping it to fit a certain amount of space
- Loading the images directly into an `ImageView` of your choice, even handling cases where that `ImageView` is recycled (e.g., part of a row in a `ListView`, where the user scrolled while an image for that `ImageView` was still loading, and now *another* image is destined for that same `ImageView` when the row was recycled)

The `HTTP/Picasso` sample application extends the Retrofit one to download the avatar image of the person asking the question, displaying it in the `ListView` along with the question title.

Downloading and Installing Picasso

Picasso can be downloaded as a small JAR from the aforementioned Web site. Android Studio users can just add a dependency for `com.squareup.picasso:picasso:...` for some version denoted by `...`, and it will pull down all other dependencies needed by Picasso:

```
dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit:retrofit:1.9.0'
}
```

(from HTTP/Picasso/app/build.gradle)

Updating the Model

Our original data model did not include information about the owner. Hence, we need to augment our data model, so Retrofit pulls that information out of the Stack Exchange JSON and makes it available to us.

To that end, we now have an `Owner` class, holding onto the one piece of information we need about the owner: the URL to the avatar (a.k.a., "profile image"):

```
package com.commonsware.android.picasso;

import com.google.gson.annotations.SerializedName;

public class Owner {
  @SerializedName("profile_image") String profileImage;
}
```

(from HTTP/Picasso/app/src/main/java/com/commonsware/android/picasso/Owner.java)

The JSON key for this in the Stack Exchange API is `profile_image`, and underscores are not the conventional way of separating words in a Java data member. Java samples usually use "camelCase" instead. The default behavior of Retrofit would require us to name our data member `profile_image` to match the JSON.

However, under the covers, Retrofit is using Google's Gson to do the mapping from JSON to objects. Gson supports a `@SerializedName` annotation, to indicate the JSON key to use for this data member. This allows us to give the data member the more natural name of `profileImage`, by using `@SerializedName("profile_image")` to teach Gson how to populate it properly.

(The author would like to thank Alec Holmes for his assistance with the Gson support)

Our `Item` class now has an `Owner`, named `owner`, since the owner data is in the `owner` key of an item's JSON object:

```
package com.commonsware.android.picasso;

public class Item {
  String title;
  Owner owner;
  String link;

  @Override
```

```
  public String toString() {
    return(title);
  }
}
```

(from HTTP/Picasso/app/src/main/java/com/commonsware/android/picasso/Item.java)

Those two changes are sufficient for Retrofit to give us our URL to be able to download the image.

Requesting the Images

Using Picasso is extremely simple, as it offers a fluent interface that allows us to set up a request in a single Java statement.

The statement begins with a call to the static `with()` method on the `Picasso` class, where we supply a `Context` (such as our activity) for Picasso to use. The statement *ends* with a call to `into()`, indicating the `ImageView` into which Picasso should load an image. In between those calls, we can chain other calls, as `with()` and most other methods on a `Picasso` object return the `Picasso` object itself.

So, we can do something like:

```
      Picasso.with(getActivity()).load(item.owner.profileImage)
             .fit().centerCrop()
             .placeholder(R.drawable.owner_placeholder)
             .error(R.drawable.owner_error).into(icon);
```

(from HTTP/Picasso/app/src/main/java/com/commonsware/android/picasso/QuestionsFragment.java)

Here, we:

- Indicate that we want to `load()` an image found at a certain URL, identified by the `profileImage` data member of the `Owner` inside an `Item` referred to as `item`
- Say that we want to `fit()` the image to our target `ImageView`
- Specify that the image should be resized using `centerCrop()` rules, to center the image within the desired size (if it is smaller on one or both axes) and to crop the image (if it is larger on one or both axes)
- Indicate that we want to put a certain drawable resource as the `placeholder()` image to show in the `ImageView` while the loading is going on in the background

- State that we want to show a certain drawable resource in the `ImageView` in case of an `error()` when the image was being loaded

And that's it. Picasso will go off, download the image, and pour it into the `ImageView` when it is ready (and resized).

The Rest of the Story

That bit of Picasso code is in a new `getView()` method on our `ItemsAdapter`:

```
class ItemsAdapter extends ArrayAdapter<Item> {
  ItemsAdapter(List<Item> items) {
    super(getActivity(), R.layout.row, R.id.title, items);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    Item item=getItem(position);
    ImageView icon=(ImageView)row.findViewById(R.id.icon);

    Picasso.with(getActivity()).load(item.owner.profileImage)
           .fit().centerCrop()
           .placeholder(R.drawable.owner_placeholder)
           .error(R.drawable.owner_error).into(icon);

    TextView title=(TextView)row.findViewById(R.id.title);

    title.setText(Html.fromHtml(getItem(position).title));

    return(row);
  }
}
```

(from HTTP/Picasso/app/src/main/java/com/commonsware/android/picasso/QuestionsFragment.java)

We have created our own row layout (`res/layout/row.xml`), consisting of an `ImageView` and a `TextView`. We have `ArrayAdapter` inflate or recycle our row, retrieve the `Item` for this row, retrieve the `ImageView` out of the row, use Picasso to start loading the real image, fill in the HTML-entity-aware text into the `TextView`, and then return our updated row. By the time we return the row, Picasso will have already loaded the placeholder image, which is what the user will initially see, while we download the real image.

The result is that we now have icons next to each of our question titles:

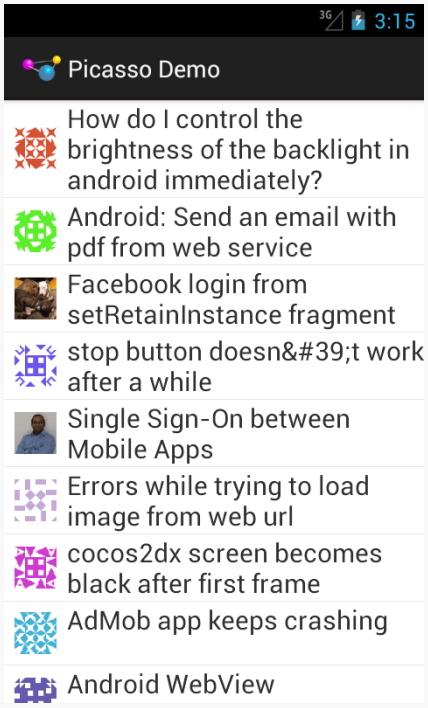


Figure 301: The Picasso Demo App

Volley

At the Google I|O 2013 conference, there was a session about Volley, an HTTP client library created by Google and used by internal apps, such as the Play Store. Volley can be thought of as a superset of Retrofit plus Picasso, minus Picasso's non-HTTP image loading facilities.

On the plus side, Volley *is* such a superset and therefore a single code base can be used to replace multiple libraries. Also, given Volley's use by Google, one imagines that this code has been applied to the widest range of possible devices. And, in early 2016, Volley finally started being distributed as an artifact that we can add to our Android Studio projects via a simple `implementation` statement.

However, there is no documentation beyond that I|O video and a training module. There is no support mechanism, except perhaps via ad-hoc social media inquiries and general support sites (e.g., Stack Overflow).

All that being said, Volley is still rather popular, so let's see how one can use it in an Android project. Specifically, the `HTTP/Volley` sample application is based off of the

Picasso sample, migrated over to use Volley instead of the combination of Picasso and Retrofit.

Getting Volley

It used to be that half the battle was just getting Volley in the first place. For years, Google only gave us a dump of source code in the Android Open Source Project, rather than a proper artifact.

However, in early 2016, Google quietly released a Volley artifact, so you can add it to your `dependencies`:

```
dependencies {
  implementation 'com.android.volley:volley:1.0.0'
}
```

Requests and Queues

Volley's primary API is via a class called `RequestQueue`. As the name suggests, it queues requests, whether those requests are for images, strings, JSON structures, or whatever. A request – in the form of a `Request` instance – embodies the URL to be retrieved, any additional information (e.g., extra HTTP headers), and the rules for interpreting the response received from the server.

Under the covers, `RequestQueue` maintains a thread pool for processing those requests. You can optionally configure this thread pool, indicating how many threads it should have and so on. You can also optionally configure how responses should be cached and the actual HTTP stack to be used for doing the network I/O. By default, on modern versions of Android, Volley delegates to `HttpUrlConnection`.

Making a Manager

Retrofit and Picasso manage application-level thread pools and caches for you, via supplied singletons. Alas, Volley does not. While this provides flexibility, it does mean that you need your own singleton wrapper around a `RequestQueue`. In the sample project, we have `VolleyManager` for that:

```
package com.commonsware.android.volley;

import android.content.Context;
import android.widget.ImageView;
```

```
import com.android.volley.Request;
import com.android.volley.RequestQueue;
import com.android.volley.toolbox.ImageLoader;
import com.android.volley.toolbox.Volley;

public class VolleyManager {
  private static volatile VolleyManager INSTANCE;
  private final RequestQueue queue;
  private final ImageLoader imageLoader;

  synchronized static VolleyManager get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=new VolleyManager(ctxt.getApplicationContext());
    }

    return(INSTANCE);
  }

  private VolleyManager(Context ctxt) {
    queue=Volley.newRequestQueue(ctxt);
    imageLoader=new ImageLoader(queue, new LruBitmapCache(ctxt));
  }

  void enqueue(Request<?> request) {
    queue.add(request);
  }

  void loadImage(String url, ImageView iv,
                 int placeholderDrawable, int errorDrawable) {
    imageLoader.get(url,
      ImageLoader.getImageListener(iv, placeholderDrawable,
        errorDrawable));
  }
}
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/VolleyManager.java)

`VolleyManager` holds the singleton instance in a `static` field named `INSTANCE`. That is lazy-initialized via the `get()` method to retrieve the instance, as we need a `Context` for a bit of our work.

The `private` constructor creates a new `RequestQueue` via the `static` `newRequestQueue()` helper method on the `Volley` class. If you prefer to have more control (e.g., supply your own response cache), you can also create a `RequestQueue` via a constructor, then call `start()` on the `RequestQueue` to kick off the thread pool.

We will discuss the rest of `VolleyManager`, including the mysterious `ImageLoader` we are creating in the constructor, a bit later in our examination of the sample app.

Requesting JSON

Volley has built-in support for retrieving strings and images from a Web service. If the Web service serves JSON, Volley also has built-in support for parsing that JSON. However, it uses the legacy `org.json` classes from the Android SDK, which work, but are slow and clunky.

Retrofit has built-in support for retrieving a JSON payload from a Web service via an HTTP request, using Gson to parse that payload into your desired POJOs. Alas, Volley does not offer Gson support out of the box.

The author of Volley, Ficus Kirkpatrick, wrote a `GsonRequest` class that handles this and published it via a GitHub gist. This has since been replicated in the Android documentation. Since this more closely matches what we had in Retrofit, the sample project uses this `GsonRequest`:

```
// from https://gist.github.com/ficusk/5474673

package com.commonsware.android.volley;

import com.google.gson.Gson;
import com.google.gson.JsonSyntaxException;

import com.android.volley.AuthFailureError;
import com.android.volley.NetworkResponse;
import com.android.volley.ParseError;
import com.android.volley.Request;
import com.android.volley.Response;
import com.android.volley.Response.ErrorListener;
import com.android.volley.Response.Listener;
import com.android.volley.toolbox.HttpHeaderParser;

import java.io.UnsupportedEncodingException;
import java.util.Map;

/**
 * Volley adapter for JSON requests that will be parsed into Java objects by Gson.
 */
public class GsonRequest<T> extends Request<T> {
  private final Gson gson = new Gson();
  private final Class<T> clazz;
  private final Map<String, String> headers;
  private final Listener<T> listener;

  /**
   * Make a GET request and return a parsed object from JSON.
   *
   * @param url URL of the request to make
```

```
   * @param clazz Relevant class object, for Gson's reflection
   * @param headers Map of request headers
   */
  public GsonRequest(String url, Class<T> clazz, Map<String, String> headers,
                     Listener<T> listener, ErrorListener errorListener) {
    super(Method.GET, url, errorListener);
    this.clazz = clazz;
    this.headers = headers;
    this.listener = listener;
  }

  @Override
  public Map<String, String> getHeaders() throws AuthFailureError {
    return headers != null ? headers : super.getHeaders();
  }

  @Override
  protected void deliverResponse(T response) {
    listener.onResponse(response);
  }

  @Override
  protected Response<T> parseNetworkResponse(NetworkResponse response) {
    try {
      String json = new String(
        response.data, HttpHeaderParser.parseCharset(response.headers));
      return Response.success(
        gson.fromJson(json, clazz), HttpHeaderParser.parseCacheHeaders(response));
    } catch (UnsupportedEncodingException e) {
      return Response.error(new ParseError(e));
    } catch (JsonSyntaxException e) {
      return Response.error(new ParseError(e));
    }
  }
}
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/GsonRequest.java)

A custom `Request` implementation like this needs four things:

1. To supply the HTTP method, URL, and a `Response.ErrorListener` to the superclass constructor
2. To override `getHeaders()`, returning a `Map` of HTTP headers to inject into the request, or `null` if there are none
3. To override `parseNetworkResponse()`, taking the raw data from the server and turning into a `Response` wrapped around the actual data to be returned to the app (in this case, the custom POJO parsed by Gson)
4. To override `deliverResponse()`, which may be just a matter of calling `onResponse()` on a `Response.Listener`

However, many developers can get away with using just one of the built-in `Request` implementations, or perhaps other pre-built implementations, like this `GsonRequest`.

Our `QuestionsFragment` can now create a `GsonRequest` for retrieving our Stack Overflow questions and turning them into an `SOQuestions` object. In this version of the sample, the work to populate the `ListView` has been moved from `onCreateView()` to `onViewCreated()`:

```
@Override
public void onViewCreated(View view,
                          Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  GsonRequest<SOQuestions> request=
    new GsonRequest<SOQuestions>(getString(R.string.url),
      SOQuestions.class, null, this, this);

  VolleyManager.get(getActivity()).enqueue(request);
}
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/QuestionsFragment.java)

The `R.string.url` value points to a string resource containing the full URL, as it is a bit long for a Java code listing, or even to reproduce easily here in this book. However, note two things about it:

1. In the Retrofit-powered examples, we added the `tagged` query parameter via a `@Query` annotated parameter on our `StackOverflowInterface`. That was mostly to demonstrate using such parameters, as we were always requesting the `android` questions. Since we do not have Retrofit's URL construction anymore, the URL in the string resource contains the `tagged=android` query parameter.
2. Because values resource files are XML files, all the `&` characters in the URL need to be escaped as `&`, to satisfy XML parsing rules.

The rest of the `GsonRequest` parameters are the class for the response (`SOQuestions.class`), our extra HTTP headers (`null`), and our implementations of `Listener` and `ErrorListener` (both the fragment itself).

We then retrieve our `VolleyManager` singleton and call `enqueue()` on it, which in turn calls `add()` on the `RequestQueue`, to cause one of Volley's threads to go do the HTTP work and process the result.

`ErrorListener` requires an `onErrorResponse()` method, which works much like its Retrofit counterpart:

```
@Override
public void onErrorResponse(VolleyError error) {
  Toast.makeText(getActivity(), error.getMessage(),
                 Toast.LENGTH_LONG).show();
  Log.e(getClass().getSimpleName(),
        "Exception from Volley request to StackOverflow", error);
}
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/QuestionsFragment.java)

`Listener` requires an `onResponse()` method, which takes the type of data we are trying to load as a parameter (`SOQuestions`). Once again, this works like its Retrofit counterpart:

```
@Override
public void onResponse(SOQuestions questions) {
  setListAdapter(new ItemsAdapter(questions.items));
}
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/QuestionsFragment.java)

Requesting Images

Volley has an `ImageRequest`, one that works like `GsonRequest`, except that it gives you a `Bitmap` back. You are welcome to use this, particularly for occasional one-off requests for images.

However, if you are going to be fetching a lot of images — particularly in something like a `ListView`, as the sample app does — you need some more smarts than that. Picasso supplies those smarts to you automatically. With Volley, you use an `ImageLoader`. An `ImageLoader` coordinates loading many images, dealing with things like:

- canceling requests when views get recycled
- having a memory cache for images, to supplement the disk cache that Volley uses for responses
- trying to minimize redraws of the UI when multiple images are decoded roughly simultaneously
- and so on

Now, for reasons that are not entirely clear, while `ImageLoader` needs a memory cache to work properly, it does not have one. Nor does Volley itself. Instead, you have to make your own implementation of `ImageLoader.ImageCache`.

The Android documentation supplies an `LruBitmapCache` that does this, on the back of Android's `LruCache`:

```
// from http://developer.android.com/training/volley/request.html

package com.commonsware.android.volley;

import android.content.Context;
import android.graphics.Bitmap;
import android.support.v4.util.LruCache;
import android.util.DisplayMetrics;
import com.android.volley.toolbox.ImageLoader.ImageCache;

public class LruBitmapCache extends LruCache<String, Bitmap>
  implements ImageCache {

  public LruBitmapCache(int maxSize) {
    super(maxSize);
  }

  public LruBitmapCache(Context ctx) {
    this(getCacheSize(ctx));
  }

  @Override
  protected int sizeOf(String key, Bitmap value) {
    return value.getRowBytes() * value.getHeight();
  }

  @Override
  public Bitmap getBitmap(String url) {
    return get(url);
  }

  @Override
  public void putBitmap(String url, Bitmap bitmap) {
    put(url, bitmap);
  }

  // Returns a cache size equal to approximately three screens worth of images.
  public static int getCacheSize(Context ctx) {
    final DisplayMetrics displayMetrics = ctx.getResources().
      getDisplayMetrics();
    final int screenWidth = displayMetrics.widthPixels;
    final int screenHeight = displayMetrics.heightPixels;
    // 4 bytes per pixel
    final int screenBytes = screenWidth * screenHeight * 4;
```

```
    return screenBytes * 3;
  }
}
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/LruBitmapCache.java)

The key for any cache, particularly a memory cache, is its maximum size. `LruBitmapCache` lets you specify that directly if you wish. Alternatively, you can provide it with a `Context`, and it will size the cache to be the size of three full-screen images. This is not a great cache sizing algorithm — it is better to tie the size to the maximum heap size of your app — but it is what Google used.

`VolleyManager` creates an `ImageLoader` in its constructor, providing it with an `LruBitmapCache`. `VolleyManager` also has a `loadImage()` method that works a bit like the builder methods on Picasso. It takes four parameters:

- The URL of the image to load
- The `ImageView` into which to load the image
- A drawable resource ID for an image placeholder
- A drawable resource ID for an image to show if there is an error retrieving the real image (e.g., 404 from the Web server when requesting the URL)

`loadImage()` in turn passes those on to the `ImageLoader`, which will handle retrieving the image, caching it, and putting it in the supplied `ImageView`.

The `getView()` method on `ItemsAdapter` can then replace its Picasso code with a call to `loadImage()`:

```
class ItemsAdapter extends ArrayAdapter<Item> {
  ItemsAdapter(List<Item> items) {
    super(getActivity(), R.layout.row, R.id.title, items);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    Item item=getItem(position);
    ImageView icon=(ImageView)row.findViewById(R.id.icon);

    VolleyManager
      .get(getActivity())
      .loadImage(item.owner.profileImage, icon,
        R.drawable.owner_placeholder,
```

```
          R.drawable.owner_error);

      TextView title=(TextView)row.findViewById(R.id.title);

      title.setText(Html.fromHtml(getItem(position).title));

      return(row);
    }
  }
```

(from HTTP/Volley/app/src/main/java/com/commonsware/android/volley/QuestionsFragment.java)

Comparison with Retrofit + Picasso

Volley's big claim to fame is that it is used in the Play Store app and elsewhere in Google's proprietary apps, supposedly.

However, its lack of official packaging and support makes it a bit more difficult for the average developer to use. It also lacks some of the "creature comforts" of Retrofit and Picasso, requiring a few extra classes to do what Square's libraries provide directly.

Other Candidate Libraries

There are plenty of other libraries that similarly try to help simplify Android HTTP operations, including:

- AndroidAsync
- android-json-rpc
- Universal Image Loader (UIL), which will be used in some samples later in this book

If you happen to be using `support-v4` or `support-v13` from the Android Support package in your app, you might also consider Ion.

The Android Arsenal has categories for general HTTP clients/networking libraries for REST client libraries, and for image loading libraries.

Visit the Trails

As noted earlier, there is a chapter on SSL that you should read, if you run into trouble using SSL in Android or want to improve your security further than you get with just stock SSL handling.

There is also a chapter on miscellaneous network capabilities – the coverage of `DownloadManager` can be found there.

Intents, Intent Filters

We have seen `Intent` objects briefly, in our discussion of having multiple activities in our application. However, we really did not dive into too much of the details about those `Intent` objects, and they can be used in other ways besides starting up an activity. In this chapter, we will examine `Intent` and their filters.

What's Your Intent?

When Sir Tim Berners-Lee cooked up the Hypertext Transfer Protocol — HTTP – he set up a system of verbs plus addresses in the form of URLs. The address indicated a resource, such as a Web page, graphic, or server-side program. The verb indicated what should be done: GET to retrieve it, POST to send form data to it for processing, etc.

An `Intent` is similar, in that it represents an action plus context. There are more actions and more components to the context with `Intent` than there are with HTTP verbs and resources, but the concept is still the same.

Just as a Web browser knows how to process a verb+URL pair, Android knows how to find activities or other application logic that will handle a given `Intent`.

Pieces of Intents

The two most important pieces of an `Intent` are the action and what Android refers to as the "data". These are almost exactly analogous to HTTP verbs and URLs — the action is the verb, and the "data" is a `Uri`, such as `https://commonsware.com` representing an HTTP URL to some balding guy's Web site. Actions are constants, such as `ACTION_VIEW` (to bring up a viewer for the resource) or `ACTION_EDIT` (to edit the resource).

If you were to create an `Intent` combining `ACTION_VIEW` with a content `Uri` of `https://commonsware.com`, and pass that `Intent` to Android via `startActivity()`, Android would know to find and open an activity capable of viewing that resource.

There are other criteria you can place inside an `Intent`, besides the action and "data" `Uri`, such as:

1. Categories. Your "main" activity will be in the `LAUNCHER` category, indicating it should show up on the launcher menu. Other activities will probably be in the `DEFAULT` category, though other categories exist and are used on occasion.
2. A MIME type, indicating the type of resource you want to operate on.
3. A component, which is to say, the class of the activity that is supposed to receive this `Intent`.
4. "Extras", which is a `Bundle` of other information you want to pass along to the receiver with the `Intent`, that the recipient might want to take advantage of. What pieces of information a given recipient can use is up to the recipient and (hopefully) is well-documented.

You will find rosters of the standard actions, categories, and extras in the Android SDK documentation for the `Intent` class.

Intent Routing

As noted above, if you specify the target component in your `Intent`, Android has no doubt where the `Intent` is supposed to be routed to — it will launch the named activity. This might be OK if the target recipient (e.g., the activity to be started) is in your application. It definitely is not recommended for invoking functionality in other applications. Component names, by and large, are considered private to the application and are subject to change. Actions, `Uri` templates, and MIME types are the preferred ways of identifying capabilities you wish third-party code to supply.

If you do not specify the target component, then Android has to figure out what recipients are eligible to receive the `Intent`. For example, Android will take the `Intent` you supply to `startActivity()` and find the activities that might support it. Note the use of the plural "activities", as a broadly-written intent might well resolve to several activities. That is the... ummm... intent (pardon the pun), as you will see later in this chapter. This routing approach is referred to as implicit routing.

Basically, there are three rules, all of which must be true for a given activity to be eligible for a given `Intent`:

- The activity must support the specified action
- The activity must support the stated MIME type (if supplied)
- The activity must support all of the categories named in the `Intent`

The upshot is that you want to make your `Intent` specific enough to find the right recipient, and no more specific than that.

This will become clearer as we work through some examples throughout this chapter.

Stating Your Intent(ions)

All Android components that wish to be started via an `Intent` must declare `Intent` filters, so Android knows which intents should go to that component. A common approach for this is to add one or more `<intent-filter>` elements to your `AndroidManifest.xml` file, inside the element for the component that should respond to the `Intent`.

For example, all of the sample projects in this book have an `<intent-filter>` on an `<activity>` that looks like this:

```
<intent-filter>
  <action android:name="android.intent.action.MAIN"/>
  <category android:name="android.intent.category.LAUNCHER"/>
</intent-filter>
```

Here, we declare that this activity:

1. Is the main activity for this application
2. It is in the `LAUNCHER` category, meaning it gets an icon in anything that thinks of itself as a "launcher", such as the home screen

You are welcome to have more than one action or more than one category in your `Intent` filters. That indicates that the associated component (e.g., activity) handles multiple different sorts of `Intent` patterns.

Responding to Implicit Intents

We saw in the chapter on multiple activities how one activity can start another via an explicit `Intent`, identifying the particular activity to be started:

```
startActivity(new Intent(this, OtherActivity.class));
```

In that case, `OtherActivity` does not need an `<intent-filter>` in the manifest. It will automatically respond when somebody explicitly identifies it as the desired activity.

However, what if you want to respond to an implicit `Intent`, one that focuses on an action string and other values? Then you *will* need an `<intent-filter>` in the manifest.

For example, take a look at the `Intents/FauxSender` sample project.

Here, we have an activity, `FauxSender`, set up to respond to an `ACTION_SEND` `Intent`, specifically for content that has the MIME type of `text/plain`:

```
    <activity
      android:name="FauxSender"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

(from Intents/FauxSender/app/src/main/AndroidManifest.xml)

The call to `startActivity()` will always add the `DEFAULT` category if no other category is specified, which is why our `<intent-filter>` also filters on that category.

Hence, if somebody on the system calls `startActivity()` on an `ACTION_SEND` `Intent` with a MIME type of `text/plain`, our `FauxSender` activity might get control. We will explain the use of the term "might" in the next section.

The documentation for `ACTION_SEND` indicates that a standard extra on the `Intent` is `EXTRA_TEXT`, representing the text to be sent. There might also be an `EXTRA_SUBJECT`, representing a subject line, if the "send" operation might have such a concept, such as an email client.

`FauxSender` can retrieve those extras and make use of them:

```
package com.commonsware.android.fsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class FauxSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String msg=getIntent().getStringExtra(Intent.EXTRA_TEXT);

    if (TextUtils.isEmpty(msg)) {
      msg=getIntent().getStringExtra(Intent.EXTRA_SUBJECT);
    }

    if (TextUtils.isEmpty(msg)) {
      Toast.makeText(this, R.string.no_message_supplied,
                     Toast.LENGTH_LONG).show();
    }
    else {
      Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
    }

    finish();
  }
}
```

(from Intents/FauxSender/app/src/main/java/com/commonsware/android/fsender/FauxSender.java)

Here, we use `TextUtils.isEmpty()` to detect if an extra is either `null` or has an empty string as its value. If `EXTRA_TEXT` is supplied, we show it in a `Toast`. Otherwise, we use `EXTRA_SUBJECT` if it is supplied, and if that is also missing, we show a stock message from a string resource.

The activity then immediately calls `finish()` from `onCreate()` to get rid of itself. That, coupled with `android:theme="@android:style/Theme.Translucent.NoTitleBar"` in the `<activity>` element, means that the activity will have no user interface, beyond the `Toast`. If run from the launcher, you will still see the launcher behind the `Toast`:

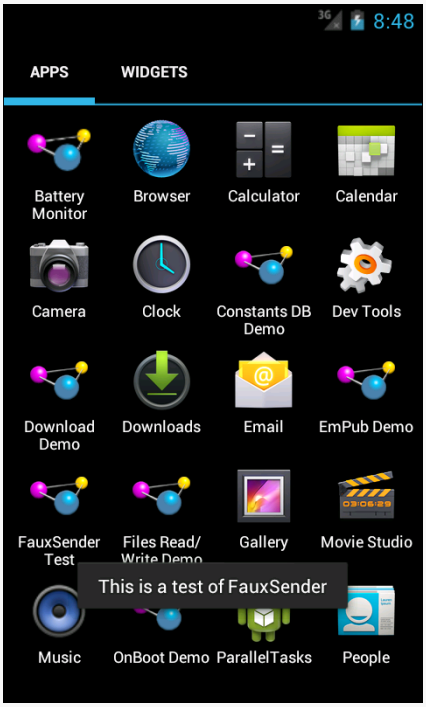


Figure 302: FauxSender, Showing EXTRA_TEXT

Requesting Implicit Intents

To send something via `ACTION_SEND`, you first set up the `Intent`, containing whatever information you want to send in `EXTRA_TEXT`, such as this code from the `FauxSenderTest` activity:

```
    Intent i=new Intent(Intent.ACTION_SEND);

    i.setType("text/plain");
    i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
    i.putExtra(Intent.EXTRA_TEXT, theMessage);
```

(from Intents/FauxSender/app/src/main/java/com/commonsware/android/fsender/FauxSenderTest.java)

(where `theMessage` is a passed-in parameter to the method containing this code fragment)

If we call `startActivity()` on this `Intent` directly, there are three possible outcomes, described in the following sections.

Zero Matches

It is possible, though unlikely, that there are no activities at all on the device that will be able to handle this `Intent`. In that case, we crash with an `ActivityNotFoundException`. This is a `RuntimeException`, which is why we do not have to keep wrapping all our `startActivity()` calls in `try/catch` blocks. However, if we might start something that does not exist, we really should catch that exception... or avoid the call in the first place. Detecting up front whether there will be any matches for our activity is a topic that will be discussed later in this book.

Note that the odds of an `ActivityNotFoundException` climb substantially on Android 4.3+ tablets, when a restricted profile is in use, as will be discussed later in this book.

One Match

It is possible that there will be exactly one matching activity. In that case, the activity in question starts up and takes over the foreground. This is what we see with the explicit `Intent`.

Many Matches, Default Behavior

It is possible that there will be more than one matching activity. In that case, by default, the user will be presented with a so-called “chooser” dialog box:

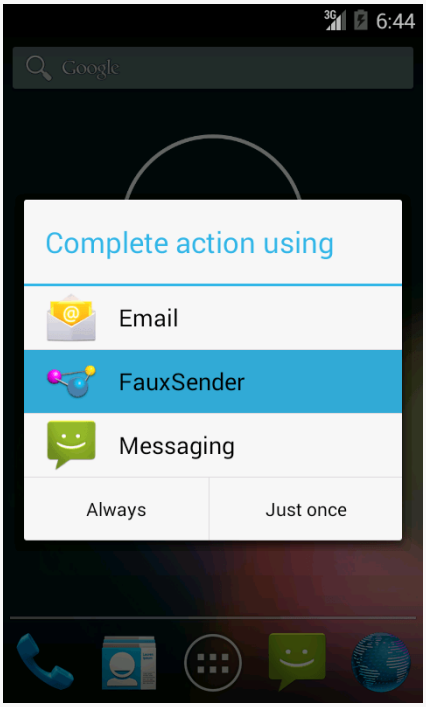


Figure 303: Chooser Dialog

The user can tap on any item in the list to have that particular activity be the one to process this event. And, if the user clicks on "Always", and we invoke the same basic `Intent` again (same action, same MIME type, same categories, same `Uri` scheme), whatever the user chooses *now* will be used again automatically, bypassing the chooser. The "Always" button in the chooser dialog sets the default activity for handling the particular `Intent` structure that triggered the chooser.

The Chooser Override

For many `Intent` patterns, the notion of the user choosing a default makes perfect sense. For example, if the user installs another Web browser, until they set a default activity, every time they go to view a Web page, they will be presented with a chooser, to choose among the installed browsers. This can get annoying quickly.

However, `ACTION_SEND` is one of those cases where a default activity is usually inappropriate. Just because the user on Monday chose to send something via Bluetooth and accidentally clicked "Always" does not mean that every day thereafter, they *always* want *every* `ACTION_SEND` to go via Bluetooth, instead of Gmail or Email or Facebook or Twitter or any other `ACTION_SEND`-capable apps they may have installed.

You can elect to force a chooser to display, regardless of whether the user has set a default activity or not. To do this, instead of calling `startActivity()` on the `Intent` directly, you wrap the `Intent` in another `Intent` returned by the `createChooser()` static method on `Intent` itself:

```
void sendIt(String theMessage) {
  Intent i=new Intent(Intent.ACTION_SEND);

  i.setType("text/plain");
  i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
  i.putExtra(Intent.EXTRA_TEXT, theMessage);

  startActivity(Intent.createChooser(i,
                                     getString(R.string.share_title)));
}
```

(from Intents/FauxSender/app/src/main/java/com/commonsware/android/fsender/FauxSenderTest.java)

The second parameter to `createChooser()` is a message to appear at the top of the dialog box:

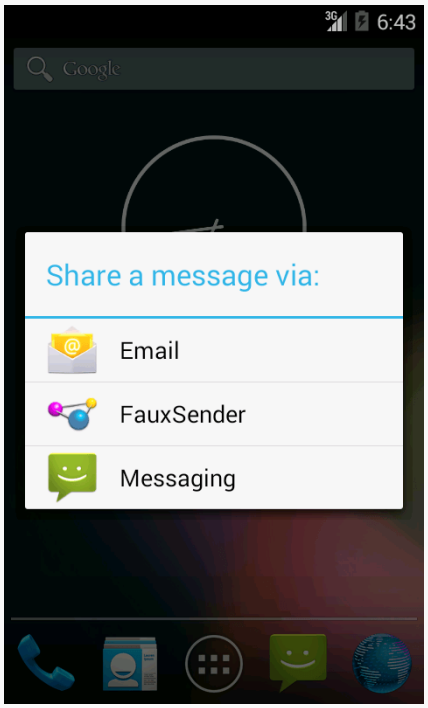


Figure 304: Your Tailored Chooser Dialog

Notice the lack of the "Always" button — not only must the user make a choice now, but also they cannot make a default choice for the future, either.

Direct Share Targets

On Android 6.0, it is possible for an app to not only have an activity appear in the chooser, but also to provide "direct share targets". These items in the chooser are supplied by the app offering an `ACTION_SEND` implementation, but rather than representing a simple activity, they can also have additional data in the extras `Bundle` in the `Intent` used to start that activity. The idea is that the app could offer a few icons to allow sharing to some fine-grained destination, such as a particular contact or a particular folder or something.

Direct share targets are covered later in this book.

ShareActionProvider

Above, we saw how you can bring up a chooser when using `startActivity()` on an implicit `Intent` action, such as `ACTION_SEND`.

There is another option, if you are using the action bar: `ShareActionProvider`. Designed for use with `ACTION_SEND`, `ShareActionProvider` supplies a drop-down menu in the action bar to let the user invoke some implementation of an `Intent` that you configure and supply.

To see how you can add a `ShareActionProvider` to your activity or fragment, let us take a look at the `ActionBar/ShareNative` sample project.

Our activity — `MainActivity` — will utilize the action bar. Its action bar items are contained in a `res/menu/actions.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/share"
    android:actionProviderClass="android.widget.ShareActionProvider"
    android:showAsAction="ifRoom"/>

</menu>
```

(from ActionBar/ShareNative/app/src/main/res/menu/actions.xml)

In addition to specifying an ID and indicating that the item should be shown in the action bar if there is room, we also include the `android:actionProviderClass` attribute. This points to a concrete implementation of the `ActionProvider` abstract base class, which is responsible for rendering the action bar item. In our case, we are using `ShareActionProvider`.

Our activity UI is simply a large `EditText` widget:

```
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/editor"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="left|top"
  android:inputType="textMultiLine"/>
```

(from ActionBar/ShareNative/app/src/main/res/layout/activity_main.xml)

We load that layout in `onCreate()` of `MainActivity`, along with initializing an `Intent` to be used when we employ the `ShareActionProvider`:

```
package com.commonsware.android.sap;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.view.Menu;
import android.widget.EditText;
import android.widget.ShareActionProvider;
import android.widget.Toast;

public class MainActivity extends Activity implements
    ShareActionProvider.OnShareTargetSelectedListener, TextWatcher {
  private ShareActionProvider share=null;
  private Intent shareIntent=new Intent(Intent.ACTION_SEND);
  private EditText editor=null;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.activity_main);

    shareIntent.setType("text/plain");
    editor=(EditText)findViewById(R.id.editor);
    editor.addTextChangedListener(this);
```

```
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    share=
        (ShareActionProvider)menu.findItem(R.id.share)
                                 .getActionProvider();
    share.setOnShareTargetSelectedListener(this);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onShareTargetSelected(ShareActionProvider source,
                                       Intent intent) {
    Toast.makeText(this, intent.getComponent().toString(),
                   Toast.LENGTH_LONG).show();

    return(false);
  }

  @Override
  public void afterTextChanged(Editable s) {
    shareIntent.putExtra(Intent.EXTRA_TEXT, s.toString());
    share.setShareIntent(shareIntent);
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start, int count,
                                int after) {
    // ignored
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                            int count) {
    // ignored
  }
}
```

(from ActionBar/ShareNative/app/src/main/java/com/commonsware/android/sap/MainActivity.java)

We also register the activity itself to be a `TextWatcher`, to find out when the user types something into the `EditText` widget.

`onCreateOptionsMenu()` is where we configure the `ShareActionProvider`, which we obtain by calling `findItem()` on our `Menu` to get the item associated with the provider, then calling `getActionProvider()` on the supplied `MenuItem`. Specifically:

- We supply an `Intent` — configured with the action, MIME type, etc. that we wish to invoke — to `setShareIntent()`
- We supply `MainActivity` itself, as an implementation of `OnShareTargetSelectedListener`, via `setOnShareTargetSelectedListener()`

In the `afterTextChanged()` method needed by the `TextWatcher` interface, we update the `EXTRA_TEXT` extra in the `Intent` to be the current contents of the `EditText`. This way, as the user types, we keep the `Intent` "fresh" with respect to what should be shared. Many consumers of a `ShareActionProvider` will have less dynamic contents, in which case you can just set up the `Intent` up front before you register it with the `ShareActionProvider`.

If the user chooses an item from the `ShareActionProvider`, we are notified via a call to our `onShareTargetSelected()` method. Registering as the `OnShareTargetSelectedListener` is optional — Android will automatically start the selected activity without our involvement. `onShareTargetSelected()` is there if you wish to know the means of sharing that the user chose. In our case, we just flash a `Toast` to indicate that the callback worked.

Practice Safe Content Resolution

NOTE: the following is based on a blog post from the author.

Dominik Schürmann and Lars Wolf, in an excellent blog post and pre-pub paper, points out a security flaw in many activities that have an `ACTION_SEND` `<intent-filter>`.

Many `ACTION_SEND` implementations accept `EXTRA_STREAM` as input. That is supposed to point to `Uri` representing a stream of stuff to be sent somewhere. Email apps might send it as an attachment, for example. So, you fire up a `ContentResolver`, call `openInputStream()` to get at the content backed by that `Uri` (because that's how real developers do it), and then do something with that content. You might not even really care what the content *is*... until that content is something from your own app. Like, say, your user account database.

The problem outlined in Mr. Schürmann's post and paper is that a malicious party could provide you with a `file:` `Uri` *to your own internal storage*. While the third-party app cannot access your internal storage, you can. So, in the case of an email app, the attacker asks you to email one of your app's own files to the attacker's email address. The user may be involved in this (e.g., having to actually click something to send the email), but with a bit of phishing or social engineering, that problem can be handled, at least some of the time. After all, courtesy of the `intent:` scheme, some Web browsers and the like will allow a simple link click to trigger the evil `ACTION_SEND` request.

To help with this, cketti (of K-9 Mail fame) wrote a `SafeContentResolver` that has its own `openInputStream()` method. However, this one will fail if the `Uri` points to a file that your app owns or to a `ContentProvider` from your app. If you use this instead of the `openInputStream()` on `ContentResolver`, your `ACTION_SEND` implementation will be safer from this attack.

More generally, if you accept input from outside parties, **validate it**. Have rules for what sorts of `Uri` values you will and will not accept for things like `EXTRA_STREAM`, and provide runtime checks to confirm that the values you receive follow the rules.

Broadcasts and Broadcast Receivers

One channel of the `Intent` message bus is used to start activities. A second channel of the `Intent` message bus is used to send broadcasts. As the name suggests, a broadcast `Intent` is one that — by default – is published to any and all applications on the device that wish to tune in.

Sending a Simple Broadcast

The simplest way to send a broadcast `Intent` is to create the `Intent` you want, then call `sendBroadcast()`.

That's it.

At that point, Android will scan through everything set up to tune into a broadcast matching your `Intent`, typically filtering just on the action string. Anyone set up to receive this broadcast will, indeed, receive it, using a `BroadcastReceiver`.

Receiving a Broadcast: In an Activity

To receive such a broadcast in an activity (or a fragment), you will need to do four things.

First, you will need to create an instance of your own subclass of `BroadcastReceiver`. The only method you need to (or should) implement is `onReceive()`, which will be passed the `Intent` that was broadcast, along with a `Context` object that, in this case, you will typically ignore.

Second, you will need to create an instance of an `IntentFilter` object, describing the sorts of broadcasts you want to receive. Most of these filters are set up to watch for a single broadcast `Intent` action, in which case the simple constructor suffices:

```
new IntentFilter(Intent.ACTION_CAMERA_BUTTON)
```

Third, you will need to call `registerReceiver()`, typically from `onStart()` of your activity or fragment, supplying your `BroadcastReceiver` and your `IntentFilter`.

Fourth, you will need to call `unregisterReceiver()`, typically from `onStop()` of your activity or fragment, supplying the same `BroadcastReceiver` instance you provided to `registerReceiver()`.

In between the calls to `registerReceiver()` and `unregisterReceiver()`, you will receive any broadcasts matching the `IntentFilter`.

The biggest downside to this approach is that some activity has to register the receiver. Sometimes, you want to receive broadcasts even when there is no activity around. To do that, you will need to use a different technique: registering the receiver in the manifest.

Receiving a Broadcast: Via the Manifest

You can also tell Android about broadcasts you wish to receive by adding a `<receiver>` element to your manifest, identifying the class that implements your `BroadcastReceiver` (via the `android:name` attribute), plus an `<intent-filter>` that describes the broadcast(s) you wish to receive:

```
<receiver android:name=".OnBootReceiver">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

The good news is that this `BroadcastReceiver` will be available for broadcasts occurring at any time. There is no assumption that you have an activity already running that called `registerReceiver()`.

The bad news is that the instance of the `BroadcastReceiver` used by Android to process a broadcast will live for only so long as it takes to execute the `onReceive()` method. At that point, the `BroadcastReceiver` is discarded. Hence, it is not safe for a manifest-registered `BroadcastReceiver` to do anything that needs to run after

`onReceive()` itself completes, such as forking a thread. After all, Android may well terminate the process within milliseconds, if there is no other running component in the process.

More bad news: `onReceive()` is called on the main application thread — the same main application thread that handles the UI of all of your activities. And, you are subject to the same limitations as are your activity lifecycle methods and anything else called on the main application thread:

- Any time spent in `onReceive()` will freeze your UI, if you happen to have a foreground activity
- If you spend too long in `onReceive()`, Android will terminate your `BroadcastReceiver` without waiting for `onReceive()` to complete

This makes using a manifest-registered `BroadcastReceiver` a bit tricky. If the work to be done is very quick, just implement it in `onReceive()`. Otherwise, you will probably need to pair this `BroadcastReceiver` with a component known as an `IntentService`, which we will examine in the next chapter.

The Stopped State

On Android 3.1 and higher, when your app is first installed on the device, it is in a "stopped" state. This has nothing to do with `onStop()` of any activity. While in the stopped state, your manifest-registered `BroadcastReceivers` will not receive any broadcasts.

Getting Out of the Stopped State

To get out of the stopped state, something on the device, such as another app (that itself is not in the stopped state), must use an explicit `Intent` to invoke one of your components.

The most common way this happens is for the user to tap on a launcher icon associated with your launcher activity. Under the covers, the home screen's launcher will create an explicit `Intent`, identifying your activity, and use that with `startActivity()`. This moves you out of the stopped state.

Getting Into the Stopped State

As noted above, you start off in the stopped state. Once you are moved out of the stopped state, via the explicit `Intent`, you will remain out of the stopped state until one of two things happens:

1. The user uninstalls your app
2. The user "force-stops" your app

The latter normally occurs when the user clicks the "Force Stop" button on your app's screen in the Settings app (Settings > Apps). There is some evidence that some device manufacturers have tied their own device's task manager to do a "force stop" when the user removes a task — this was not a particularly wise choice on the part of those manufacturers.

Note that a reboot does *not* move you back into the stopped state. You remain in the normal state through a reboot.

Example System Broadcasts

There are many, many broadcasts sent out by Android itself, which you can tune into if you see fit. Many, but not all, of these are documented on the `Intent` class. The values in the "Constants" table that have "Broadcast Action" leading off their description are action strings used for system broadcasts. There are other such broadcast actions scattered around the SDK, though, so do not assume that they are all documented on `Intent`.

The following sections will examine two of these broadcasts, to see how the `BroadcastReceiver` works in action.

At Boot Time

A popular request is to have code get control when the device is powered on. This is doable but somewhat dangerous, in that too many on-boot requests slow down the device startup and may make things sluggish for the user.

In order to be notified when the device has completed its system boot process, you will need to request the `RECEIVE_BOOT_COMPLETED` permission. Without this, even if you arrange to receive the boot broadcast `Intent`, it will not be dispatched to your receiver.

As the Android documentation describes it:

> Though holding this permission does not have any security implications, it can have a negative impact on the user experience by increasing the amount of time it takes the system to start and allowing applications to have themselves running without the user being aware of them. As such, you must explicitly declare your use of this facility to make that visible to the user.

We also need to register our `BroadcastReceiver` in the manifest — by the time an activity would call `registerReceiver()`, the boot will have long since occurred.

For example, let us examine the `Intents/OnBoot` sample project.

In our manifest, we request the needed permission and register our `BroadcastReceiver`, along with an activity:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sysevents.boot"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver android:name=".OnBootReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>

    <activity
      android:name="BootstrapActivity"
```

```
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Intents/OnBoot/app/src/main/AndroidManifest.xml)

`OnBootReceiver` simply logs a message to Logcat:

```
package com.commonsware.android.sysevents.boot;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Log.d(getClass().getSimpleName(), "Hi, Mom!");
  }
}
```

(from Intents/OnBoot/app/src/main/java/com/commonsware/android/sysevents/boot/OnBootReceiver.java)

To test this on Android 3.0 and earlier, simply install the application and reboot the device — you will see the message appear in Logcat.

However, on Android 3.1 and higher, the user must first manually launch some activity before any manifest-registered `BroadcastReceiver` objects will be used, as noted above in the section covering the stopped state. Hence, if you were to just install the application and reboot the device, nothing would happen. The little `BootstrapActivity` is merely there for the user to launch, so that the `ACTION_BOOT_COMPLETED` `BroadcastReceiver` will start working.

On Battery State Changes

One theme with system events is to use them to help make your users happier by reducing your impacts on the device while the device is not in a great state. Most

applications are impacted by battery life. Dead batteries run no apps. Hence, knowing the battery level may be important for your app.

There is an `ACTION_BATTERY_CHANGED` `Intent` that gets broadcast as the battery status changes, both in terms of charge (e.g., 80% charged) and charging (e.g., the device is now plugged into AC power). You simply need to register to receive this `Intent` when it is broadcast, then take appropriate steps.

One of the limitations of `ACTION_BATTERY_CHANGED` is that you have to use `registerReceiver()` to set up a `BroadcastReceiver` to get this `Intent` when broadcast. You cannot use a manifest-declared receiver. There are separate `ACTION_BATTERY_LOW` and `ACTION_BATTERY_OK` broadcasts that you *can* receive from a manifest-registered receiver, but they are broadcast far less frequently, only when the battery level falls below or rises above some undocumented "low" threshold.

To demonstrate `ACTION_BATTERY_CHANGED`, take a peek at the `Intents/OnBattery` sample project.

In there, you will find a `res/layout/batt.xml` resource containing a `ProgressBar`, a `TextView`, and an `ImageView`, to serve as a battery monitor:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ProgressBar
    android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
      android:id="@+id/level"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:textSize="36sp"/>
```

```
    <ImageView
      android:id="@+id/status"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>

</LinearLayout>
```

(from Intents/OnBattery/app/src/main/res/layout/batt.xml)

This layout is used by a `BatteryFragment`, which registers to receive the `ACTION_BATTERY_CHANGED` `Intent` in `onStart()` and unregisters in `onStop()`:

```
package com.commonsware.android.battmon;

import android.app.Fragment;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.BatteryManager;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.ProgressBar;
import android.widget.TextView;

public class BatteryFragment extends Fragment {
  private ProgressBar bar=null;
  private ImageView status=null;
  private TextView level=null;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.batt, parent, false);

    bar=(ProgressBar)result.findViewById(R.id.bar);
    status=(ImageView)result.findViewById(R.id.status);
    level=(TextView)result.findViewById(R.id.level);

    return(result);
  }
```

```
@Override
public void onStart() {
  super.onStart();

  IntentFilter f=new IntentFilter(Intent.ACTION_BATTERY_CHANGED);

  getActivity().registerReceiver(onBattery, f);
}

@Override
public void onStop() {
  getActivity().unregisterReceiver(onBattery);

  super.onStop();
}

BroadcastReceiver onBattery=new BroadcastReceiver() {
  public void onReceive(Context context, Intent intent) {
    int pct=
        100 * intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 1)
            / intent.getIntExtra(BatteryManager.EXTRA_SCALE, 1);

    bar.setProgress(pct);
    level.setText(String.valueOf(pct));

    switch (intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1)) {
      case BatteryManager.BATTERY_STATUS_CHARGING:
        status.setImageResource(R.drawable.charging);
        break;

      case BatteryManager.BATTERY_STATUS_FULL:
        int plugged=
            intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);

        if (plugged == BatteryManager.BATTERY_PLUGGED_AC
            || plugged == BatteryManager.BATTERY_PLUGGED_USB) {
          status.setImageResource(R.drawable.full);
        }
        else {
          status.setImageResource(R.drawable.unplugged);
        }
        break;

      default:
        status.setImageResource(R.drawable.unplugged);
        break;
    }
  }
```

```
  };
}
```

(from Intents/OnBattery/app/src/main/java/com/commonsware/android/battmon/BatteryFragment.java)

The key to `ACTION_BATTERY_CHANGED` is in the "extras". Many extras are packaged in the `Intent`, to describe the current state of the battery, such as the following constants defined on the `BatteryManager` class:

- `EXTRA_HEALTH`, which should generally be `BATTERY_HEALTH_GOOD`
- `EXTRA_LEVEL`, which is the proportion of battery life remaining as an integer, specified on the scale described by the `EXTRA_SCALE` value
- `EXTRA_PLUGGED`, which will indicate if the device is plugged into AC power (`BATTERY_PLUGGED_AC`) or USB power (`BATTERY_PLUGGED_USB`)
- `EXTRA_SCALE`, which indicates the maximum possible value of level (e.g., 100, indicating that level is a percentage of charge remaining)
- `EXTRA_STATUS`, which will tell you if the battery is charging (`BATTERY_STATUS_CHARGING`), full (`BATTERY_STATUS_FULL`), or discharging (`BATTERY_STATUS_DISCHARGING`)
- `EXTRA_TECHNOLOGY`, which indicates what sort of battery is installed (e.g., `"Li-Ion"`)
- `EXTRA_TEMPERATURE`, which tells you how warm the battery is, in tenths of a degree Celsius (e.g., 213 is 21.3 degrees Celsius)
- `EXTRA_VOLTAGE`, indicating the current voltage being delivered by the battery, in millivolts

In the case of `BatteryFragment`, when we receive an `ACTION_BATTERY_CHANGED` `Intent`, we do three things:

1. We compute the percentage of battery life remaining, by dividing the level by the scale
2. We update the `ProgressBar` and `TextView` to display the battery life as a percentage
3. We display an icon, with the icon selection depending on whether we are charging (status is `BATTERY_STATUS_CHARGING`), full but on the charger (status is `BATTERY_STATUS_FULL` and plugged is `BATTERY_PLUGGED_AC` or `BATTERY_PLUGGED_USB`), or are not plugged in

If you plug this into a device, it will show you the device's charge level:

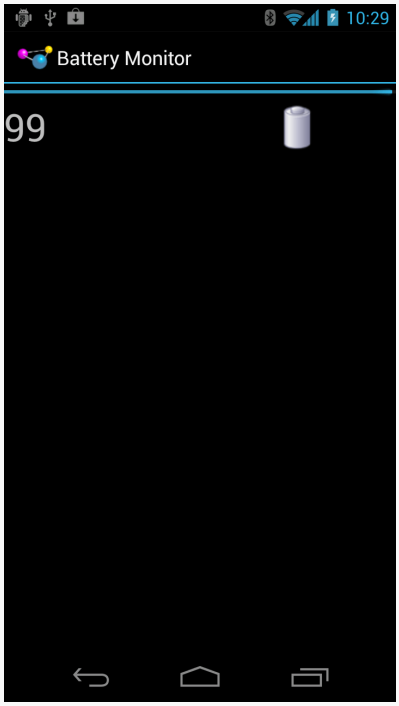


Figure 305: The Battery Monitor

Sticky Broadcasts and the Battery

NOTE: Sticky broadcasts are deprecated in Android 5.0, and the documentation hints that they may be abandoned entirely in the future.

Android has a notion of "sticky broadcast `Intents`". Normally, a broadcast `Intent` will be delivered to interested parties and then discarded. A sticky broadcast `Intent` is delivered to interested parties and retained until the next matching `Intent` is broadcast. Applications can call `registerReceiver()` with an `IntentFilter` that matches the sticky broadcast, but with a `null` `BroadcastReceiver`, and get the sticky `Intent` back as a result of the `registerReceiver()` call.

This may sound confusing. Let's look at this in the context of the battery.

Earlier in this section, you saw how to register for `ACTION_BATTERY_CHANGED` to get information about the battery delivered to you. You can also, though, get the latest battery information without registering a receiver. Just create an `IntentFilter` to match `ACTION_BATTERY_CHANGED` (as shown above) and call `registerReceiver()` with that filter and a `null` `BroadcastReceiver`. The `Intent` you get back from `registerReceiver()` is the last `ACTION_BATTERY_CHANGED` `Intent` that was broadcast,

with the same extras. Hence, you can use this to get the current (or near-current) battery status, rather than having to bother registering an actual `BroadcastReceiver`.

This is why the sample app shows its results immediately — it was given the last-broadcast edition of the `ACTION_BATTERY_CHANGED` broadcast once we called `registerReceiver()`.

Battery and the Emulator

Your emulator does not really have a battery. If you run this sample application on an emulator, you will see, by default, that your device has 50% fake charge remaining and that it is being charged. However, it is charged infinitely slowly, as it will not climb past 50%... at least, not without help.

NOTE: At the time of this writing, the Linux emulator does not properly emulate the battery for AVDs created from certain device profiles (e.g., Nexus S), showing 0% battery charge and not responding to the `telnet` commands described below. If you encounter this, go into the `config.ini` file for your AVD (found in `~/.android/avd/.../`, where `~/` is your home directory and `...` is the name of the AVD) and add `hw.battery=yes` as a property. If that property exists but is set to `no`, change it to `yes`.

While the emulator will only show fixed battery characteristics, you can change what those values are, through the highly advanced user interface known as `telnet`.

You may have noticed that your emulator title bar consists of the name of your AVD plus a number, frequently 5554. That number is not merely some engineer's favorite number. It is also an open port, on your emulator, to which you can `telnet` into, on `localhost` (127.0.0.1) on your development machine.

There are many commands you can issue to the emulator by means of `telnet` . To change the battery level, use `power capacity NN`, where `NN` is the percentage of battery life remaining that you wish the emulator to return. If you do that while you have an `ACTION_BATTERY_CHANGED` `BroadcastReceiver` registered, the receiver will receive a broadcast `Intent`, informing you of the change.

You can also experiment with some of the other power subcommands (e.g., `power ac on` or `power ac off`), or other commands (e.g., `geo`, to send simulated GPS fixes, just as you can do from DDMS).

Battery Data on Android 5.0+

As noted earlier, Android 5.0 deprecates sticky broadcasts. The existing broadcasts still work, though. And, even if someday Android gets rid of sticky broadcasts entirely, broadcasts like `ACTION_BATTERY_CHANGED` most likely will *still* work, albeit just as a regular broadcast.

To get current battery information on Android 5.0 and higher, `BatteryManager` offers `getIntProperty()` and `getLongProperty()`, where the keys for the "properties" are `BATTERY_PROPERTY_*` constants defined on `BatteryManager`, such as `BATTERY_PROPERTY_CAPACITY` to determine the percentage of remaining battery capacity.

The Order of Things

Another variation on the broadcast `Intent` is the ordered broadcast.

Normally, if you broadcast an `Intent`, and there are 10 registered `BroadcastReceivers` that match that `Intent`, all 10 will receive the broadcast, in indeterminate order, and possibly in parallel (particularly on multi-core devices).

With an ordered broadcast, the behavior shifts a bit:

- Only one `BroadcastReceiver` at a time will receive the broadcast
- The order in which the `BroadcastReceivers` receive the broadcast is (somewhat) controlled by their developers
- A `BroadcastReceiver` can "abort" the broadcast, preventing other receivers in the chain from receiving it

Sending an ordered broadcast is merely a matter of calling `sendOrderedBroadcast()`.

Receiving an ordered broadcast, at its core, is identical to receiving a regular broadcast: you write a `BroadcastReceiver` and register it via the manifest or `registerReceiver()`. However, you have two additional options when registering that `BroadcastReceiver`.

First, you can specify a priority, either via `setPriority()` on the `IntentFilter` or `android:priority` on the `<intent-filter>` element. The priority is an integer, with higher numbers indicating higher priority. Higher-priority receivers will get the

broadcast sooner than will lower-priority receivers. The default priority is `0`. In theory, the priority should be a value between `-1000` and `1000`, but this does not seem to be checked by the system, and many apps use a priority higher than `1000`.

Second, your `BroadcastReceiver` can call `abortBroadcast()` to consume the event, preventing any lower-priority receivers from even seeing the broadcast.

Keeping It Local

A broadcast `Intent`, by default and nearly by definition, is broadcast. Anything on the device could have a receiver "tuned in" to listen for such broadcasts. While you can use `setPackage()` on `Intent` to restrict the distribution, the broadcast still goes through the standard broadcast mechanism, which involves transferring the `Intent` to an OS process, which then does the actual broadcasting. Hence, a broadcast `Intent` has some overhead.

Yet, there are times when using broadcasts within an app is handy, but it would be nice to avoid the overhead. To help with this the core Android team added `LocalBroadcastManager` to the Android Support package, to provide an in-process way of doing broadcasts with the standard `Intent`, `IntentFilter`, and `BroadcastReceiver` classes, yet with less overhead.

`LocalBroadcastManager` is supplied by both the `android-support-v4.jar` and `android-support-v13.jar` libraries. Generally speaking, if your `android:minSdkVersion` is less than 13, you probably should choose `android-support-v4.jar`.

The only real difference, from a coding standpoint, in using `LocalBroadcastManager` is that you call `registerReceiver()`, `unregisterReceiver()`, and `sendBroadcast()` on an instance of `LocalBroadcastManager`, instead of on an instance of `Context`. You get the `LocalBroadcastManager` singleton for your process via a static `getInstance()` method on `LocalBroadcastManager` itself.

We will see `LocalBroadcastManager` in use in one of the samples in the services chapter.

Visit the Trails!

We examine `LocalBroadcastManager` in more detail, along with other event bus alternatives, later in the book.

Tutorial #15 - Sharing Your Notes

Perhaps you would like to get your notes off of our book reader app and into someplace else, or perhaps you would like to share them with somebody else. Either way, we can do that using an `ACTION_SEND` operation, to allow the user to choose how to "send" the notes, such as sending them by email or uploading them to some third-party note service.

To make this work, we will add a `ShareActionProvider` to our action bar on the `NoteFragment`.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Adding a ShareActionProvider

First, we need to allow the user to indicate that they want to "share" the note displayed in the current `NoteFragment`. By putting an action bar item on the activity where the `NoteFragment` is displayed, we do not need to worry about letting the user choose which note to send — we simply send whichever note they happen to be viewing or editing.

By using a `ShareActionProvider`, the action item will handle most of the work for allowing the user to choose where to send the note to. We only need to provide an `Intent` that identifies what is to be shared.

Modify `res/menu/notes.xml` to add in the new `share` toolbar button:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/share"
    android:actionProviderClass="android.widget.ShareActionProvider"
    android:showAsAction="ifRoom"
    android:title="@string/share"/>
  <item
    android:id="@+id/delete"
    android:icon="@drawable/ic_delete_white_24dp"
    android:showAsAction="ifRoom|withText"
    android:title="@string/delete">
  </item>
</menu>
```

(from EmPubLite-AndroidStudio/T15-Share/EmPubLite/app/src/main/res/menu/notes.xml)

Note that this menu definition requires a new string resource, named `share`, with a value like `Share`.

Step #2: Sharing the Note

Now, we need to configure the `ShareActionProvider`, in particular supplying it with a continuously-updated `Intent`, based upon what the user has typed into the `EditText`.

Add a `ShareActionProvider` data member to `NoteFragment`, named `share`, along with an `Intent` data member named `shareIntent` configured to use `ACTION_SEND` of a MIME type of `text/plain`:

```
  private ShareActionProvider share=null;
  private Intent shareIntent=
    new Intent(Intent.ACTION_SEND).setType("text/plain");
```

(from EmPubLite-AndroidStudio/T15-Share/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Then, in `onCreateView()`, tell the `EditText` to let us know when the user changes the text, via `addTextChangedListener()`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);
```

```
    editor=(EditText)result.findViewById(R.id.editor);
    editor.addTextChangedListener(this);

    return(result);
  }
```

(from EmPubLite-AndroidStudio/T15-Share/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

This will fail to compile, as our `NoteFragment` is not implementing the `TextWatcher` interface. So, modify the `NoteFragment` class declaration to include the `TextWatcher` interface:

```
public class NoteFragment extends Fragment implements TextWatcher {
```

(from EmPubLite-AndroidStudio/T15-Share/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

That, in turn, will require us to implement three methods:

1. `afterTextChanged()`
2. `beforeTextChanged()`
3. `onTextChanged()`

In our case, we care about `afterTextChanged()`. So, add the following three methods to `NoteFragment`:

```
  @Override
  public void afterTextChanged(Editable s) {
    shareIntent.putExtra(Intent.EXTRA_TEXT, s.toString());
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start, int count,
                                int after) {
    // ignored
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                            int count) {
    // ignored
  }
```

(from EmPubLite-AndroidStudio/T15-Share/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Here, we update the `shareIntent` with the latest text to be shared, storing it in `EXTRA_TEXT`, per the instructions in the Android developer documentation for working with `ACTION_SEND`.

However, we have not initialized `share` yet. We can do that in `onCreateOptionsMenu()`, adding a call to `findItem()` to find our `R.id.share` menu item, then calling `getActionProvider()` to get the `ShareActionProvider` out of the menu item:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.notes, menu);

  share=
    (ShareActionProvider)menu.findItem(R.id.share)
      .getActionProvider();
  share.setShareIntent(shareIntent);

  super.onCreateOptionsMenu(menu, inflater);
}
```

(from EmPubLite-AndroidStudio/T15-Share/EmPubLite/app/src/main/java/com/commonsware/empublite/NoteFragment.java)

Here, we also attach the `shareIntent` to the `ShareActionProvider`, so when it comes time to share the text, the `ShareActionProvider` knows how to do that.

Step #3: Testing the Result

If you run this on a device and navigate to a filled-in note, you will see the new action bar item:

Figure 306: ShareActionProvider in NoteFragment

If you tap on it, you will get a roster of possible ways to share the text:

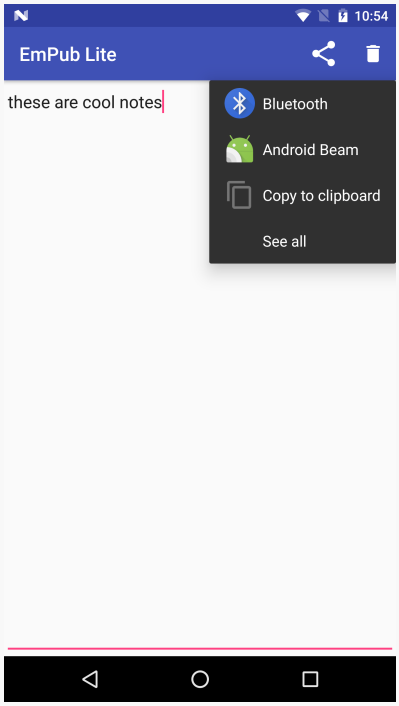


Figure 307: ShareActionProvider in NoteFragment, Expanded

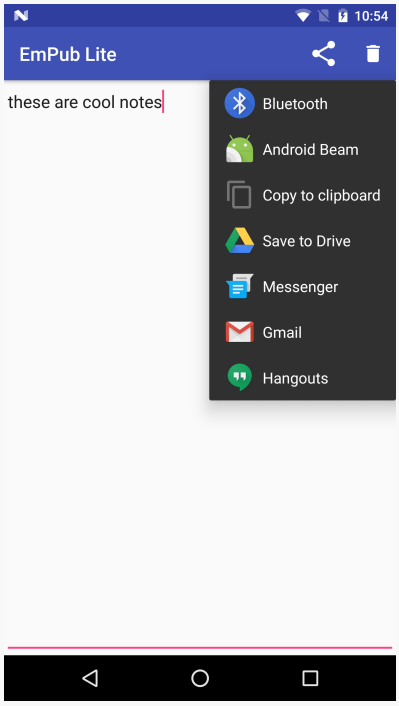


Figure 308: ShareActionProvider in NoteFragment, Fully Expanded

The exact options you see will vary based on your device or emulator, and what apps are installed on it that know how to share plain text. If you only have one choice (e.g., Messenger), it will appear next to the share icon, and you will only be able to tap on that one choice.

Unfortunately, your emulator may have nothing that can handle this `Intent`. If that is the case, you will crash with an `ActivityNotFoundException`. To get past this, if you enter `http://goo.gl/w113e` in your emulator's browser, that should allow you to download and install a copy of the APK from the `Intents/FauxSender` sample project that we covered earlier in this book. When the download is complete (which should be very quick), open up the notification drawer and tap on the "download complete" notification. This should begin the installation process. Depending on your Android version, you may also need to "allow installation of non-Market apps" — after fixing this, you can use the Downloads app on the emulator to try installing the APK again. Once FauxSender is installed, it will respond to your attempts to share a note.

In Our Next Episode…

… we will allow the user to update the book's contents over the Internet.

Services and the Command Pattern

As noted previously, Android services are for long-running processes that may need to keep running even when decoupled from any activity. Examples include playing music even if the "player" activity is destroyed, polling the Internet for RSS/Atom feed updates, and maintaining an online chat connection even if the chat client loses focus due to an incoming phone call.

Services are created when manually started (via an API call) or when some activity tries connecting to the service via inter-process communication (IPC). Services will live until specifically shut down, until Android is desperate for RAM and terminates the process, or for a short period of time on Android 8.0+. Running for a long time has its costs, though, so services need to be careful not to use too much CPU or keep radios active too much of the time, lest the service cause the device's battery to get used up too quickly.

This chapter outlines the basic theory behind creating and consuming services, including a look at the "command pattern" for services.

Why Services?

Services are a "Swiss Army knife" for a wide range of functions that do not require direct access to an activity's user interface, such as:

1. Performing operations that need to continue even if the user leaves the application's activities, like a long download (as seen with the Play Store) or playing music (as seen with Android music apps)
2. Performing operations that need to exist regardless of activities coming and going, such as maintaining a chat connection in support of a chat application

3. Providing a local API to remote APIs, such as might be provided by a Web service
4. Performing periodic work without user intervention, akin to cron jobs or Windows scheduled tasks

Even things like home screen app widgets often involve a service to assist with long-running work.

The primary role of a service is as a flag to the operating system, letting it know that your process is still doing work, despite the fact that it is in the background. This makes it *somewhat* less likely that Android will terminate your process due to low memory conditions.

Many applications will not need any services. Very few applications will need more than one. However, the service is a powerful tool for an Android developer's toolbox and is a subject with which any qualified Android developer should be familiar.

Setting Up a Service

Creating a service implementation shares many characteristics with building an activity. You inherit from an Android-supplied base class, override some lifecycle methods, and hook the service into the system via the manifest.

The Service Class

There are many service classes that you might inherit from.

The root of all of them is `Service`, just as the root of the hierarchy of activity classes is `Activity`. You may wish to subclass `Service`, particularly if:

- You will be using the binding pattern for communicating to that service from another app or process, or
- The service will need to run for an indeterminate period of time, such as a media player that works via user control

Historically, the next-most-common base class was `IntentService`. This was good for a "transactional" bit of work, where you need to have a service do something in the background for a bit, then you no longer need the service. The classic example here is a service to download a large file: once the file is downloaded, you no longer need that service to be around. `IntentService` supplies you with a background

thread to use for the network I/O and disk I/O, and it will shut down automatically once the work is complete.

However, `IntentService` has recently been supplanted by `JobIntentService`. This has similar characteristics, but it works better on Android 8.0+ devices, due to limitations on what you can do in the background.

We will examine all three of these solutions in this chapter.

Beyond those, there are many specialized service classes. Elsewhere in the book you will find examples of:

- `JobService`, for doing periodic or scheduled work
- `TileService`, for adding a custom tile to the notification shade of the device
- `ChooserTargetService`, for helping provide more specific integration options with other apps

And there are plenty of others. While they are all services, and they will all be added to the manifest in the same fashion (more or less), the API that you implement will be very specific to the service that you are extending. The API for `TileService` looks little like the API for `JobService`, for example. As such, those services, where we use them, are covered elsewhere in the book.

Lifecycle Methods

Just as activities have `onCreate()`, `onResume()`, `onPause()` and kin, `Service` implementations have their own lifecycle methods, such as:

- `onCreate()`, which, as with activities, is called when the service is created, by any means
- `onStartCommand()`, which is called each time the service is sent a command via `startService()`
- `onBind()`, which is called whenever a client binds to the service via `bindService()`
- `onDestroy()` which is called as the service is being shut down

As with activities, services initialize whatever they need in `onCreate()` and clean up those items in `onDestroy()`. And, as with activities, the `onDestroy()` method of a service might not be called, if Android terminates the entire application process, such as for emergency RAM reclamation.

The `onStartCommand()` and `onBind()` lifecycle methods will be implemented based on your choice of communicating to the client, as will be explained later in this chapter.

Note that `Service` is an `abstract` class and `onBind()` is an `abstract` method, so even if you are not using `bindService()`, you will need to implement `onBind()` in order to successfully compile. A common approach here is to have `onBind()` simply return `null`.

Note that these methods may be optional, depending upon your needs and the base class that you extend. So, for example, overriding any of these in a `JobIntentService` is unusual at best and a bad idea at worst.

Manifest Entry

Finally, you need to add the service to your `AndroidManifest.xml` file, for it to be recognized as an available service for use. That is simply a matter of adding a `<service>` element as a child of the `application` element, providing `android:name` to reference your service class.

Since the service class is in the same Java namespace as everything else in this application, we can use the shorthand (e.g., `"PlayerService"`) to reference our class.

For example, here is a manifest showing the `<service>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.fakeplayer"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
```

```
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name="FakePlayer"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <service android:name="PlayerService"/>
  </application>

</manifest>
```

(from Service/FakePlayer/app/src/main/AndroidManifest.xml)

Communicating To Services

Clients of services — frequently activities, though not necessarily — have two main ways to send requests or information to a service. One approach is to send a command, which creates no lasting connection to the service. The other approach is to bind to the service, establishing a communications channel that lasts as long as the client needs it.

Sending Commands with startService()

The simplest way to work with a service is to call `startService()`. The `startService()` method takes an `Intent` parameter, much like `startActivity()` does. In fact, the `Intent` supplied to `startService()` has the same two-part role as it does with `startActivity()`:

1. Identify the service to communicate with
2. Supply parameters, in the form of `Intent` extras, to tell the service what it is supposed to do

For a local service — the focus of this chapter — the simplest form of `Intent` is one that identifies the class that implements the `Service` (e.g., `new Intent(this, MyService.class);`).

The call to `startService()` is asynchronous, so the client will not block. The service will be created if it is not already running, and it will receive the `Intent` via a call to

the `onStartCommand()` lifecycle method. The service can do whatever it needs to in `onStartCommand()`, but since `onStartCommand()` is called on the main application thread, it should do its work very quickly. Anything that might take more than a handful of milliseconds should be delegated to a background thread.

The `onStartCommand()` method can return one of several values, mostly to indicate to Android what should happen if the service's process should be killed while it is running. The most likely return values are:

1. `START_STICKY`, meaning that the service should be moved back into the started state (as if `onStartCommand()` had been called), but do not re-deliver the `Intent` to `onStartCommand()`
2. `START_REDELIVER_INTENT`, meaning that the service should be restarted via a call to `onStartCommand()`, supplying the same `Intent` as was delivered this time
3. `START_NOT_STICKY`, meaning that the service should remain stopped until explicitly started by application code

By default, calling `startService()` not only sends the command, but tells Android to keep the service running until something tells it to stop. One way to stop a service is to call `stopService()`, supplying the same `Intent` used with `startService()`, or at least one that is equivalent (e.g., identifies the same class). At that point, the service will stop and will be destroyed. Note that `stopService()` does not employ any sort of reference counting, so three calls to `startService()` will result in a single service running, which will be stopped by a call to `stopService()`.

Another possibility for stopping a service is to have the service call `stopSelf()` on itself. You might do this if you use `startService()` to have a service begin running and doing some work on a background thread, then having the service stop itself when that background work is completed.

Note that `JobIntentService` has an API that feels like the command pattern, though it uses a different set of methods, as will be explored later in the chapter.

Binding to Services

Another approach to communicating with a service is to use the binding pattern. Here, instead of packaging commands to be sent via an `Intent`, you can obtain an actual API from the service, with whatever data types, return values, and so on that you wish. You then invoke that API no different than you would on some local object.

The benefit is the richer API. The cost is that binding is more complex to set up and more complex to maintain, particularly across configuration changes.

We will discuss the binding pattern later in this book.

Scenario: The Music Player

Most audio player applications in Android — for music, audiobooks, or whatever — do not require the user to remain in the player application itself. Rather, the user can go on and do other things with their device, with the audio playing in the background.

The sample project reviewed in this section is `Service/FakePlayer`.

The Design

We will use `startService()`, since we want the service to run even when the activity starting it has been destroyed. However, we will use a regular `Service`, rather than an `IntentService` or `JobIntentService`. Those are designed to do work and stop itself, whereas in this case, we want the user to be able to stop the music playback when the user wants to.

Since music playback is outside the scope of this chapter, the service will simply stub out those particular operations.

The Service Implementation

Here is the implementation of this `Service`, named `PlayerService`:

```
package com.commonsware.android.fakeplayer;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class PlayerService extends Service {
  public static final String EXTRA_PLAYLIST="EXTRA_PLAYLIST";
  public static final String EXTRA_SHUFFLE="EXTRA_SHUFFLE";
  private boolean isPlaying=false;

  @Override
```

```
  public int onStartCommand(Intent intent, int flags, int startId) {
    String playlist=intent.getStringExtra(EXTRA_PLAYLIST);
    boolean useShuffle=intent.getBooleanExtra(EXTRA_SHUFFLE, false);

    play(playlist, useShuffle);

    return(START_NOT_STICKY);
  }

  @Override
  public void onDestroy() {
    stop();
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }

  private void play(String playlist, boolean useShuffle) {
    if (!isPlaying) {
      Log.w(getClass().getName(), "Got to play()!");
      isPlaying=true;
    }
  }

  private void stop() {
    if (isPlaying) {
      Log.w(getClass().getName(), "Got to stop()!");
      isPlaying=false;
    }
  }
}
```

(from Service/FakePlayer/app/src/main/java/com/commonsware/android/fakeplayer/PlayerService.java)

In this case, we really do not need anything for `onCreate()`, so that lifecycle method is skipped. On the other hand, we have to implement `onBind()`, because that is an abstract method on `Service`.

When the client calls `startService()`, `onStartCommand()` is called in `PlayerService`. Here, we get the `Intent` and pick out some extras to tell us what to play back (`EXTRA_PLAYLIST`) and other configuration details (e.g., `EXTRA_SHUFFLE`). `onStartCommand()` calls `play()`, which simply flags that we are playing and logs a message to Logcat — a real music player would use `MediaPlayer` to start playing the first song in the playlist. `onStartCommand()` returns `START_NOT_STICKY`, indicating

that if Android terminates the process (e.g., low memory), it should not restart it once conditions improve.

`onDestroy()` stops the music from playing — theoretically, anyway — by calling a `stop()` method. Once again, this just logs a message to Logcat, plus updates our internal are-we-playing flag.

Using the Service

The `PlayerFragment` demonstrating the use of `PlayerService` has a very elaborate UI, consisting of two large buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/start"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/start_the_player"/>

  <Button
    android:id="@+id/stop"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/stop_the_player"/>

</LinearLayout>
```

(from Service/FakePlayer/app/src/main/res/layout/main.xml)

The fragment itself is not much more complex:

```
package com.commonsware.android.fakeplayer;

import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
```

```
public class PlayerFragment extends Fragment implements
    View.OnClickListener {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, parent, false);

    result.findViewById(R.id.start).setOnClickListener(this);
    result.findViewById(R.id.stop).setOnClickListener(this);

    return(result);
  }

  @Override
  public void onClick(View v) {
    Intent i=new Intent(getActivity(), PlayerService.class);

    if (v.getId() == R.id.start) {
      i.putExtra(PlayerService.EXTRA_PLAYLIST, "main");
      i.putExtra(PlayerService.EXTRA_SHUFFLE, true);

      getActivity().startService(i);
    }
    else {
      getActivity().stopService(i);
    }
  }
}
```

(from Service/FakePlayer/app/src/main/java/com/commonsware/android/fakeplayer/PlayerFragment.java)

The `onCreateView()` method merely loads the UI. The `onClick()` method constructs an `Intent` with fake values for `EXTRA_PLAYLIST` and `EXTRA_SHUFFLE`, then calls `startService()`. After you press the "Start" button, you will see the corresponding message in Logcat. Similarly, `stopPlayer()` calls `stopService()`, triggering the second Logcat message. Notably, you do not need to keep the activity running in between those button clicks — you can exit the activity via BACK and come back later to stop the service.

Communicating *From* Services

Sending commands to a service, by default, is a one-way street. Frequently, though, we need to get results from our service back to our activity. There are a few approaches for how to accomplish this.

Broadcast Intents

One approach, first mentioned in the chapter on Intent filters, is to have the service send a broadcast Intent that can be picked up by the activity... assuming the activity is still around and is not paused. The service can call sendBroadcast(), supplying an Intent that identifies the broadcast, designed to be picked up by a BroadcastReceiver. This could be a component-specific broadcast (e.g., new Intent(this, MyReceiver.class)), if the BroadcastReceiver is registered in the manifest. Or, it can be based on some action string, perhaps one even documented and designed for third-party applications to listen for.

The activity, in turn, can register a BroadcastReceiver via registerReceiver(), though this approach will only work for Intent objects specifying some action, not ones identifying a particular component. But, when the activity's BroadcastReceiver receives the broadcast, it can do what it wants to inform the user or otherwise update itself.

However, for local services, this is not a good choice. System broadcasts like this are intrinsically system-wide; for a local service, you should be using a communications channel that is private to your process.

Pending Results

Your activity can call createPendingResult(). This returns a PendingIntent – an object that represents an Intent and the corresponding action to be performed upon that Intent (e.g., use it to start an activity). In this case, the PendingIntent will cause a result to be delivered to your activity's implementation of onActivityResult(), just as if another activity had been called with startActivityForResult() and, in turn, called setResult() to send back a result.

Since a PendingIntent is Parcelable, and can therefore be put into an Intent extra, your activity can pass this PendingIntent to the service. The service, in turn, can call one of several flavors of the send() method on the PendingIntent, to notify the activity (via onActivityResult()) of an event, possibly even supplying data (in the form of an Intent) representing that event.

We will be seeing PendingIntent used many places later in this book, such as with notifications and AlarmManager.

Event Buses

Event bus implementations — like `LocalBroadcastManager` or greenrobot's EventBus — are a great solution for having a service communicate with objects elsewhere within your process. You can have the service raise events (e.g., `NewEmailEvent`, `UploadCompletedEvent`, `MartiansHaveLandedEvent`), which activities or fragments can listen for and respond to.

Messenger

Yet another possibility is to use a `Messenger` object. A `Messenger` sends messages to an activity's `Handler`. Within a single activity, a `Handler` can be used to send messages to itself, as was mentioned briefly in the chapter on threads. However, between components — such as between an activity and a service — you will need a `Messenger` to serve as the bridge.

As with a `PendingIntent`, a `Messenger` is `Parcelable`, and so can be put into an `Intent` extra. The activity calling `startService()` or `bindService()` would attach a `Messenger` as an extra on the `Intent`. The service would obtain that `Messenger` from the `Intent`. When it is time to alert the activity of some event, the service would:

1. Call `Message.obtain()` to get an empty `Message` object
2. Populate that `Message` object as needed, with whatever data the service wishes to pass to the activity
3. Call `send()` on the `Messenger`, supplying the `Message` as a parameter

The `Handler` will then receive the message via `handleMessage()`, on the main application thread, and so can update the UI or whatever is necessary.

Notifications

Another approach is for the service to let the user know directly about the work that was completed. To do that, a service can raise a `Notification` — putting an icon in the status bar and optionally shaking or beeping or something. This technique is covered in an upcoming chapter.

We can also combine these techniques, such as using an event bus event and detecting when nothing in the UI layer receives the event, so we know that we need to display a `Notification`. We will be examining this pattern later in the book as well.

Scenario: The Downloader

If you elect to download something from the Play Store, you are welcome to back out of the Play Store application entirely. This does not cancel the download – the download and installation run to completion, despite no Play Store activity being on-screen.

You may have similar circumstances in your application, from downloading a purchased e-book to downloading a map for a game to downloading a file from some sort of "drop box" file-sharing service. And, perhaps `DownloadManager` is not going to be a great choice, for any number of reasons (e.g., you want to download the file to internal storage).

The sample project reviewed in this section is `Service/Downloader`, which implements such a downloading service.

The Design

This sort of situation is a perfect use for the command pattern and either an `IntentService` or a `JobIntentService`. Either of those have a background thread, so downloads can take as long as needed. Either of those will automatically shut down when the work is done, so the service will not linger and you do not need to worry about shutting it down yourself. Your activity can simply send a command to tell it to go do the work.

In this sample, we will use an `IntentService`. A clone of this project that uses a `JobIntentService` appears later in this chapter.

Admittedly, things get a bit trickier when you want to have the activity find out when the download is complete. This example will show the use of `LocalBroadcastManager` for this.

Things get even trickier when you want to download to a public location on external storage, such as the `Downloads` directory. On Android 6.0+ devices, with a `targetSdkVersion` of 23 or higher, you need to request runtime permissions before you can write to external storage. However, requesting runtime permissions needs to be done by the UI layer — a service cannot request permissions on its own (though it can check to see if the app has permission). The simplest thing to do is to request the permissions, if needed, before starting the service. This sample app demonstrates this.

Using the Service

The `DownloadFragment` demonstrating the use of `Downloader` has a trivial UI, consisting of one large button:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/button"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="@string/do_the_download"
/>
```

(from Service/Downloader/app/src/main/res/layout/main.xml)

That UI is initialized in `onCreateView()`, as usual:

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, parent, false);

    b=result.findViewById(R.id.button);
    b.setOnClickListener(this);

    return(result);
  }
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

When the user clicks the button, `onClick()` is called. What happens now depends on whether we have permission to write to external storage or not:

```
  @Override
  public void onClick(View v) {
    if (hasPermission(WRITE_EXTERNAL_STORAGE)) {
      doTheDownload();
    }
    else {
      requestPermissions(
        new String[] { WRITE_EXTERNAL_STORAGE }, REQUEST_STORAGE);
    }
  }
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

The first time the user runs the app, the app will not have permission yet. `hasPermission(WRITE_EXTERNAL_STORAGE)` will return `false`, where `hasPermission()` is a utility method wrapping around `ContextCompat.checkSelfPermission()`:

```
  private boolean hasPermission(String perm) {
    return(ContextCompat.checkSelfPermission(getActivity(), perm)==
      PackageManager.PERMISSION_GRANTED);
  }
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

`WRITE_EXTERNAL_STORAGE` is a `static` import, just to cut down on verbosity:

```
import static android.Manifest.permission.WRITE_EXTERNAL_STORAGE;
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

If `hasPermission()` returns `false`, we call `requestPermissions()` on `FragmentCompat`, as we are in a fragment, not an activity. That eventually routes to `onRequestPermissionsResult()`:

```
  @Override
  public void onRequestPermissionsResult(int requestCode, String[] permissions,
                                         int[] grantResults) {
    if (hasPermission(WRITE_EXTERNAL_STORAGE)) {
      doTheDownload();
    }
  }
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

Hence, if we either have permission when the user clicks the button, or if we receive permission after asking for it, we call a `doTheDownload()` method to kick off the download. Specifically, we disable the button (to prevent accidental duplicate downloads) and call `startService()` to send over a command:

```
  private void doTheDownload() {
    b.setEnabled(false);

    Intent i=new Intent(getActivity(), Downloader.class);

    i.setData(Uri.parse("https://commonsware.com/Android/Android-1_0-CC.pdf"));

    getActivity().startService(i);
  }
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

Here, the `Intent` we pass over has the URL of the file to download (in this case, a URL pointing to a PDF).

The Service Implementation

Here is the implementation of this `IntentService`, named `Downloader`:

```
package com.commonsware.android.downloader;

import android.app.IntentService;
import android.content.Intent;
import android.os.Environment;
import android.support.v4.content.LocalBroadcastManager;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class Downloader extends IntentService {
  public static final String ACTION_COMPLETE=
      "com.commonsware.android.downloader.action.COMPLETE";

  public Downloader() {
    super("Downloader");
  }

  @Override
  public void onHandleIntent(Intent i) {
    try {
      File root=
          Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

      root.mkdirs();

      File output=new File(root, i.getData().getLastPathSegment());

      if (output.exists()) {
        output.delete();
      }

      URL url=new URL(i.getData().toString());
      HttpURLConnection c=(HttpURLConnection)url.openConnection();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;
```

```
        while ((len=in.read(buffer)) >= 0) {
          out.write(buffer, 0, len);
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
        c.disconnect();
      }

      LocalBroadcastManager.getInstance(this)
                           .sendBroadcast(new Intent(ACTION_COMPLETE));
    }
    catch (IOException e2) {
      Log.e(getClass().getName(), "Exception in download", e2);
    }
  }
}
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

Our business logic is in `onHandleIntent()`, which is called on an Android-supplied background thread, so we can take whatever time we need. Also, when `onHandleIntent()` ends, the `IntentService` will stop itself automatically... assuming no other requests for downloads occurred while `onHandleIntent()` was running. In that case, `onHandleIntent()` is called again for the next download, and so on.

In `onHandleIntent()`, we first set up a `File` object pointing to where we want to download the file. We use `getExternalStoragePublicDirectory()` to find the public folder for downloads. Since this directory may not exist, we need to create it using `mkdirs()`. We then use the `getLastPathSegment()` convenience method on `Uri`, which returns to us the filename portion of a path-style `Uri`. The result is that our `output` `File` object points to a file, named the same as the file we are downloading, in a public folder.

We then go through a typical `HttpUrlConnection` process to connect to the URL supplied via the `Uri` in the `Intent`, streaming the results from the connection (8KB at a time) out to our designated file. Then, we follow the requested recipe to ensure our file is saved:

- `flush()` the stream
- `sync()` the `FileDescriptor` (from `getFD()`)
- `close()` the stream

This recipe was explained back in the chapter on file I/O.

Finally, it would be nice to let somebody know that the download has completed. So, we send a local broadcast `Intent`, with our own custom action (`ACTION_COMPLETE`), using `LocalBroadcastManager`.

Note that, in theory, we could start the service, but the user could revoke our permission before we get a chance to start the download. In practice, this is very unlikely to happen, as our download should start within milliseconds. However, when working with runtime permissions from services, you need to consider the length of time between confirming that you have permission and when you perform the actions secured by that permission. If there may be a significant time gap, double-check the permission before trying the actions (e.g., writing to external storage). It is cleaner to recover from `checkSelfPermission()` indicating that you do not have permission than from some `IOException` or `SecurityException` because you do not have permission and tried the action anyway.

If your service determines that it does *not* have permission, you cannot call `requestPermissions()`, as a service is neither an activity nor a fragment. Instead, raise a notification and gracefully exit the service. The notification can direct the user somewhere in the app where you can request the permission (again) and re-try the work to be done by the service.

Receiving the Broadcast

Our `DownloadFragment` is set up to listen for that local broadcast `Intent`, by registering a local `BroadcastReceiver` in `onStart()` and unregistering it in `onStop()`:

```
@Override
public void onStart() {
  super.onStart();

  IntentFilter f=new IntentFilter(Downloader.ACTION_COMPLETE);

  LocalBroadcastManager.getInstance(getActivity())
                       .registerReceiver(onEvent, f);
}

@Override
public void onStop() {
  LocalBroadcastManager.getInstance(getActivity())
                       .unregisterReceiver(onEvent);
```

```
    super.onStop();
  }
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

The `BroadcastReceiver` itself re-enables our button, plus displays a `Toast` indicating that the download is complete:

```
  private BroadcastReceiver onEvent=new BroadcastReceiver() {
    public void onReceive(Context ctxt, Intent i) {
      b.setEnabled(true);

      Toast.makeText(getActivity(), R.string.download_complete,
                     Toast.LENGTH_LONG).show();
    }
  };
```

(from Service/Downloader/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

Note that if the user leaves the activity (e.g., BACK, HOME), the broadcast will not be received by the activity. There are other ways of addressing this, particularly combining an ordered broadcast with a `Notification`, which we will examine later in this book.

Limitation: Time

On Android 7.1 and older, this sample app works fine.

On Android 8.0 and newer, this sample app works fine... because the PDF file that it is trying to download is not that big.

However, on Android 8.0+, a started service like this can run for only one minute. After that, Android will stop the service. That will not terminate our background thread immediately, but it will make it far more likely that Android will terminate our entire process, taking our thread and download with it.

As a result, nowadays, an `IntentService` is suitable only for scenarios where you are quite certain that the work will get done in less than a minute.

One way to work around this is to make the service be a "foreground service". In the upcoming chapter on notifications, we will revisit this sample and discuss the use of `startForeground()` to make a service be a foreground service.

Another workaround is to use `JobIntentService`, as we will see in the next section.

Limitation: Staying Awake

Even if our work is certain to take less than a minute, we may run into another problem: the device might fall asleep. When the screen turns off, by default, Android will power down the CPU, suspending all running processes. This, in turn, will prevent us from downloading the file anymore.

`JobIntentService` works around this by way of a "wakelock": a request to the OS to keep the device awake, even though the screen turns off. You can use a wakelock yourself to keep the device awake, for scenarios where you need that capability but where `JobIntentService` is not a great option. We will explore wakelocks in greater detail in the chapter on `AlarmManager`.

JobIntentService

`IntentService` still works on Android 8.0, but unless you make it be a foreground service, you will be limited to ~1 minute of runtime before your service is stopped abruptly. `JobIntentService` is a wrapper around a `JobService` that offers `IntentService`-style semantics. On Android 8.0 and higher, when you tell a `JobIntentService` to do some work, it enqueues that work via `JobScheduler`. On Android 7.1 and earlier, the `JobIntentService` behaves more like a regular `IntentService`, though one that supplies a `WakeLock` for you (akin to the author's `WakefulIntentService`).

The `Service/JobIntentService` sample project is a clone of the `IntentService` sample, where we use the service to download a PDF file. The revised sample swaps out the `IntentService` with a `JobIntentService`, which is a fairly easy conversion to make.

First, we need to defend it with the `android.permission.BIND_JOB_SERVICE` permission:

```
    <service
      android:name="Downloader"
      android:permission="android.permission.BIND_JOB_SERVICE" />
```

(from Service/JobIntentService/app/src/main/AndroidManifest.xml)

This is required because, under the covers, a `JobIntentService` is really a `JobService`, and `JobService` requires this `android:permission` attribute. We will explore this more in the chapter on `JobScheduler`.

What had been `onHandleIntent()` in an `IntentService` turns into `onHandleWork()` in a `JobIntentService`:

```
package com.commonsware.android.downloader;

import android.content.Context;
import android.content.Intent;
import android.os.Environment;
import android.support.v4.app.JobIntentService;
import android.support.v4.content.LocalBroadcastManager;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class Downloader extends JobIntentService {
  public static final String ACTION_COMPLETE=
      "com.commonsware.android.downloader.action.COMPLETE";
  private static final int UNIQUE_JOB_ID=1337;

  static void enqueueWork(Context ctxt, Intent i) {
    enqueueWork(ctxt, Downloader.class, UNIQUE_JOB_ID, i);
  }

  @Override
  public void onHandleWork(Intent i) {
    try {
      File root=
          Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

      root.mkdirs();

      File output=new File(root, i.getData().getLastPathSegment());

      if (output.exists()) {
        output.delete();
      }

      URL url=new URL(i.getData().toString());
      HttpURLConnection c=(HttpURLConnection)url.openConnection();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;

        while ((len=in.read(buffer)) >= 0) {
          out.write(buffer, 0, len);
        }

        out.flush();
```

```
      }
      finally {
        fos.getFD().sync();
        out.close();
        c.disconnect();
      }

      LocalBroadcastManager.getInstance(this)
                           .sendBroadcast(new Intent(ACTION_COMPLETE));
    }
    catch (IOException e2) {
      Log.e(getClass().getName(), "Exception in download", e2);
    }
  }
}
```

(from Service/JobIntentService/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

The semantics of `onHandleWork()` are the same as `doWakefulWork()` with a `WakefulIntentService`:

- The method is called on a background thread
- While there is work to be done, a wakelock is held
- Once there is no more work to be done and the method returns, the service stops itself

Because our code is using a wakelock — by way of a support library — we need to have the `WAKE_LOCK` permission in the manifest, along with other permissions needed for our business logic:

```
  <uses-permission android:name="android.permission.WAKE_LOCK" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

(from Service/JobIntentService/app/src/main/AndroidManifest.xml)

To arrange to have the service perform some work, with `IntentService`, you would use `startService()`, with an `Intent` identifying the service (and, optionally, passing along details of the work to be done). With `JobIntentService`, you instead call a static `enqueueWork()` method defined on `JobIntentService`. This takes four parameters:

- a `Context`
- the Java `Class` object for your `JobIntentService` subclass (e.g., `Downloader.class`)

- a job ID, which will be used on Android 8.0 when using `JobScheduler` to perform the work, where the job ID needs to be unique within your app compared to anything else that is using `JobScheduler`
- an `Intent` akin to the one you would have used with `startService()` and `IntentService`

Since the second and third parameters are constants, you can create your own `enqueueWork()` method that calls the `JobService` implementation, passing along those constant values:

```
private static final int UNIQUE_JOB_ID=1337;

static void enqueueWork(Context ctxt, Intent i) {
  enqueueWork(ctxt, Downloader.class, UNIQUE_JOB_ID, i);
}
```

(from Service/JobIntentService/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

Then, your code that wishes to have the work performed calls your `enqueueWork()` method to do so:

```
private void doTheDownload() {
  b.setEnabled(false);

  Intent i=new Intent(getActivity(), Downloader.class);

  i.setData(Uri.parse("https://commonsware.com/Android/Android-1_0-CC.pdf"));

  Downloader.enqueueWork(getActivity(), i);
}
```

(from Service/JobIntentService/app/src/main/java/com/commonsware/android/downloader/DownloadFragment.java)

The `JobIntentService` can spend up to ~10 minutes in `onHandleWork()` on Android 8.0+, which is a substantial improvement over the ~1 minute a background `IntentService` has. If, however, there is a substantial chance that the work would exceed 10 minutes, use a foreground `IntentService`.

However, there is one issue with `JobIntentService`: on Android 8.0+, it may not do its work right away. On Android 8.0+, `JobIntentService` schedules its work via `JobScheduler`, and `JobScheduler` may elect to postpone that work for a bit. In cases where you really need the work to be started immediately, `JobIntentService` is not a good choice.

IntentService or JobIntentService?

Use `JobIntentService` if:

- You can live with the possible slight delay in the work being started, and
- You think that the work might take more than a minute, but surely less than 10 minutes

Use a plain `IntentService` if you are sure that the work will take less than a minute.

Use a foreground `IntentService` if either:

- You think that the work might take more than a minute, but you want to avoid the possible delayed-start of a `JobIntentService`, or
- You think that the work might take more than 10 minutes

Services and Configuration Changes

Services are not directly affected by configuration changes the way that activities are. While activities will be destroyed and recreated by default, services continue running if they were created.

Usually, services do not really care about configuration changes. However, if you have a service that *does* care, you can override `onConfigurationChanged()` in the service.

This means that you have two choices for dealing with configuration changes: override `onConfigurationChanged()` or simply re-read in the configuration information as needed. For example, suppose that you need to know the user's chosen locale, to include as information in a Web service call. If you are checking the locale on each Web service call, your service does not need to know about configuration changes. If, on the other hand, you prefer to cache the locale data, reading it in from the `Locale` class when the service is created, you will want to override `onConfigurationChanged()` and update that cache, in case the configuration change was a locale change.

When Do Services End?

One common question with services is: when do services end? If we start them, what causes them to ever stop?

While services are designed to run in the background for a while, they will not run forever. Exactly how long they will run depends on what stops them, and there are several scenarios for that.

Scenario #1: You Stop It

If you call `stopSelf()` from inside a started service, the service is stopped and destroyed. Or, if you call `stopService()` from something outside of the service — but supplying an `Intent` that identifies your service — the service is stopped and destroyed. In both cases, the service is stopped and destroyed asynchronously, so the service will be running for a bit even after the `stopSelf()` or `stopService()` call. However, the service will be stopped and destroyed fairly quickly, assuming that you are not tying up the main application thread for some reason.

As we will see in the chapter on binding and remote services, if you unbind from a bound service, and nothing else has bound to it or has started it, Android will destroy the service.

So, you can cause a service to stop by taking some positive action to stop it, much as we did with the `stopService()` call in the `FakePlayer` example earlier in this chapter.

Scenario #2: Android Terminates Your Process

The real point behind a service is to elevate your process' importance within the operating system. Android terminates low-importance processes to free up system RAM for other processes representing other apps.

However, while process importance is the primary criterion for Android to choose processes to terminate, it is not the only criterion. Process age and memory consumption are also taken into account. As a result, your process will not run forever, even with a service running.

Needless to say, when your process is terminated, your service goes with it.

Scenario #3: The User Nukes You From Orbit

Not only can Android terminate your process, but so can the user, by any number of ways:

- Using a third-party task manager
- Swiping your app off of the recent-tasks ("overview") screen
- Clicking "Force Stop" on your page in the Settings app

In the first two scenarios, if you had a running service, and if your `onStartCommand()` returns something like `START_STICKY` or `START_REDELIVER_INTENT`, eventually Android will fork a fresh process for you and restart your service. This also happens if Android terminates your process on its own due to old age.

If the user presses "Force Stop", nothing of your app will run again, until the user launches your app from the home screen or something else uses an explicit `Intent` to start one of your components.

Scenario #4: Automatically, After a Minute

All of the previous options have been a possibility for a long time, frequently since Android 1.0.

New to Android 8.0 is a time limit on background services: you can run for a minute, and then you're done. We will explore this more in the next section.

Background Service Limitations

For apps that have a `targetSdkVersion` of 26 or higher and are running on Android 8.0, background services are limited. After a short period of time — as low as one minute — any such services will be stopped and you will be unable to start new ones.

Also, even if your `targetSdkVersion` is 25 or lower, you might *still* have these limitations applied to your app. If your app appears on the Battery screen in Settings — indicating that it is using above-average power — the user will have the ability to apply these limitations to your app from there.

What Is a Background Service, Exactly?

Here, "background services" are ones that:

- Have not used something like `startForeground()` to raise themselves to foreground importance (with a `Notification`),
- Are in a process that is not showing the foreground UI at the moment, and
- Are not bound to by some other process that happens to have foreground importance

That latter scenario covers cases where your service exposes an API that is bound to by other apps or by core OS processes. This includes custom APIs implemented via AIDL and framework-supplied APIs such as those used by `JobService`, `TileService`, and so on.

Certain events, such as having your code triggered by a `Notification` `PendingIntent`, or by receiving a broadcast, will also give you a fresh window of time when background services behave normally.

What Happens?

The documentation indicates that when your process moves from the foreground to the background, or when one of the other triggers (e.g., receiving a broadcast) occurs, you have "several minutes" of normal operation. Testing suggests that by "several minutes", Google actually means "a minute or so".

At that point:

- Any outstanding services are stopped and destroyed, via `stopService()`-style functionality, and
- Any calls to `startService()` will fail

Your process is still running at this point. However, its importance is the same as if you had no service running, meaning that your process is at risk of being terminated at any point to free up system RAM.

Note that while the documentation suggests that `startService()` will throw an `IllegalStateException` when your app is ineligible to start a background service, this behavior varies. If a service calls `startService()`, no exception is thrown, and the call seems to fail quietly. If another `Context` is used — such as a `BroadcastReceiver` — you will get the exception:

```
Process: com.commonsware.android.service.ouroboros, PID: 27276
java.lang.RuntimeException: Unable to start receiver
com.commonsware.android.service.ouroboros.HackReceiver:
java.lang.IllegalStateException: Not allowed to start service Intent {
cmp=com.commonsware.android.service.ouroboros/.SecondSillyService (has extras) }: app
is in background
   at android.app.ActivityThread.handleReceiver(ActivityThread.java:3159)
   at android.app.ActivityThread.-wrap18(Unknown Source:0)
   at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1621)
   at android.os.Handler.dispatchMessage(Handler.java:102)
   at android.os.Looper.loop(Looper.java:154)
   at android.app.ActivityThread.main(ActivityThread.java:6408)
   at java.lang.reflect.Method.invoke(Native Method)
   at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:232)
   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:751)
Caused by: java.lang.IllegalStateException: Not allowed to start service Intent {
cmp=com.commonsware.android.service.ouroboros/.SecondSillyService (has extras) }: app
is in background
   at android.app.ContextImpl.startServiceCommon(ContextImpl.java:1451)
   at android.app.ContextImpl.startService(ContextImpl.java:1405)
   at android.content.ContextWrapper.startService(ContextWrapper.java:630)
   at android.content.ContextWrapper.startService(ContextWrapper.java:630)
   at
com.commonsware.android.service.ouroboros.HackReceiver.onReceive(HackReceiver.java:12)
   at android.app.ActivityThread.handleReceiver(ActivityThread.java:3152)
   at android.app.ActivityThread.-wrap18(Unknown Source:0)
   at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1621)
   at android.os.Handler.dispatchMessage(Handler.java:102)
   at android.os.Looper.loop(Looper.java:154)
   at android.app.ActivityThread.main(ActivityThread.java:6408)
   at java.lang.reflect.Method.invoke(Native Method)
   at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:232)
   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:751)
```

What Are the Alternatives?

An obvious solution is to use a foreground service. You have three options for doing this:

- The classic solution, which is to start the service via `startService()`, then have the service call `startForeground()`
- The new `getForegroundService()` method on `PendingIntent`
- The new `startForegroundService()` method on `Context`

The latter two approaches work even when your process no longer has the ability to call `startService()`. However, those methods are new to Android 8.0. Also, despite their method names, they do not actually start your service as a foreground service. Rather, they give you a short reprieve from the normal service-starting limits — your service needs to call `startForeground()` shortly after being created, or else your service will be destroyed.

A `JobService` can spend ~10 minutes processing a job, before Android will consider the service to be broken and reduce the process' priority to lower levels. Hence, using `JobScheduler` may be an option for you, particularly if the work you are trying to do is periodic in nature, where your `JobService` does the work.

As this change only affects apps with a `targetSdkVersion` higher than 25, keeping your `targetSdkVersion` at 25 or lower will avoid this behavior change.

Tutorial #16 - Updating the Book

The app is designed to ship a copy of the book's chapters as assets, so a user can just download one thing and get everything they need: book and reader.

However, sometimes books get updated. This is a bit less likely with the material being used in this tutorial, as it is rather unlikely that H. G. Wells will rise from the grave to amend *The War of the Worlds*. However, other books, such as Android developer guides written by balding guys, might be updated more frequently.

Most likely, the way you would get those updates is by updating the entire app, so you get improvements to the reader as well. However, another approach would be to be able to download an update to the book as a separate ZIP file. The reader would use the contents of that ZIP file if one has been downloaded, otherwise it will "fall back" to the copy in assets. That is the approach that we will take in this tutorial, to experiment a bit with Internet access and services. Along the way, we will use Retrofit to call a Web service (of sorts) to find out if an update is available.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Adding a Stub DownloadCheckService

There are a few pieces to our download-the-book-update puzzle:

- We need to determine if there is an update available and, if so, where we can find the ZIP file that is the update
- We need to download the update's ZIP file, which could be a fairly large file

- We need to unpack that ZIP file into internal or external storage, so that it is more easily used by the rest of our code and performs more quickly than would dynamically reading the contents out of the ZIP on the fly
- All of that needs to happen in the background from a threading standpoint
- Ideally, all of that could happen either in the foreground or the background from a UI standpoint (i.e., user manually requests an update check, or an update check is performed automatically on a scheduled basis)

To address the first puzzle piece — determining if there is an update available — we can use an `IntentService`. That makes it easy for us to do the work not only in the background from a threading standpoint, but also be able to use it either from the UI or from some sort of background-work scheduler. So, let's add a `DownloadCheckService` to our project.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Service > "Service (IntentService)" from the context menu. Fill in `DownloadCheckService` as the class name and uncheck the "helper methods" checkbox. Click Finish to generate the `DownloadCheckService` class and add an entry for you to the manifest.

Then, replace the generated implementation of `DownloadCheckService` with:

```
package com.commonsware.empublite;

import android.app.IntentService;
import android.content.Intent;

public class DownloadCheckService extends IntentService {
  public DownloadCheckService() {
    super("DownloadCheckService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
  }
}
```

Step #2: Tying the Service Into the Action Bar

To allow the user to manually request that we update the book (if an update is available), we should add a new action bar item to `EmPubLiteActivity`.

Right-click over the `res/` directory and choose New > Vector Asset from the context menu. Click the Icon button and search for the "refresh" icon:

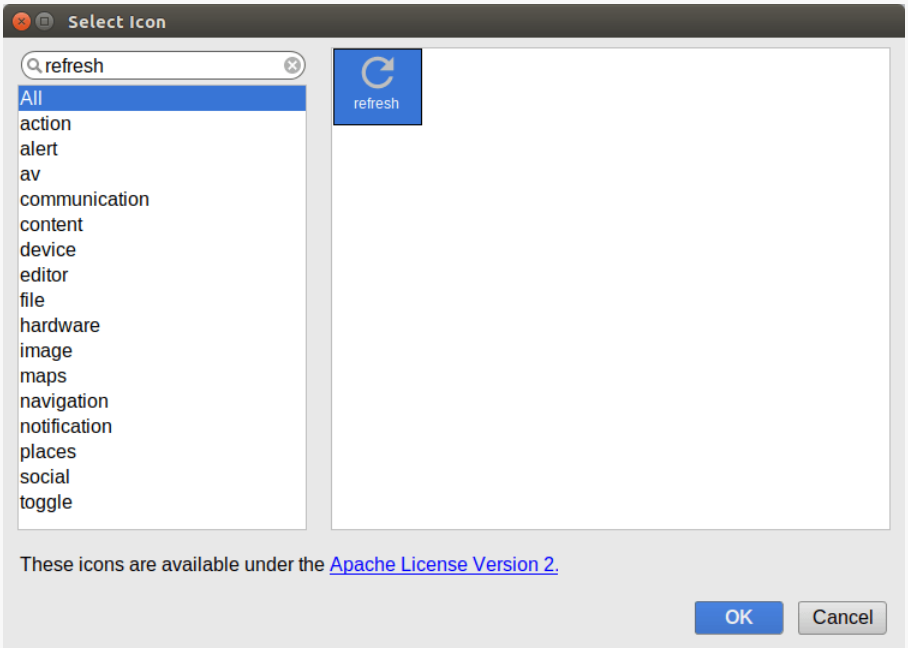


Figure 309: Asset Studio Icon Picker, with Refresh Icon Selected

Click OK to close the icon picker. Change the resource name to `ic_refresh_white_24dp`. Then click Next and Finish to save this drawable resource.

Once again, this icon will render in black, when we need it to render in white given our theme. Open `res/drawable/ic_refresh_white_24dp.xml` and change the `android:fillColor` in the `<path>` element to be `#FFFFFFFF` instead of `#FF000000`:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
        android:width="24dp"
        android:height="24dp"
        android:viewportWidth="24.0"
        android:viewportHeight="24.0">
    <path
        android:fillColor="#FFFFFFFF"
        android:pathData="M17.65,6.35C16.2,4.9 14.21,4 12,4c-4.42,0 -7.99,3.58 -7.99,8s3.57,8
7.99,8c3.73,0 6.84,-2.55 7.73,-6h-2.08c-0.82,2.33 -3.04,4 -5.65,4 -3.31,0 -6,-2.69 -6,-6s2.69,-6
6,-6c1.66,0 3.14,0.69 4.22,1.78L13,11h7V4l-2.35,2.35z"/>
</vector>
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/res/drawable/ic_refresh_white_24dp.xml)

Then, modify the `res/menu/options.xml` file to include the following `<item>` element:

```
<item
  android:id="@+id/update"
  android:icon="@drawable/ic_refresh_white_24dp"
  android:showAsAction="ifRoom|withText"
  android:title="@string/download_update">
</item>
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/res/menu/options.xml)

Note that this menu definition requires a new string resource, named `download_update`, with a value like `Download Update`.

That allows us to add a new `case` to the `switch` statement in `onOptionsItemSelected()` in `EmPubLiteActivity`:

```
    case R.id.update:
      startService(new Intent(this, DownloadCheckService.class));

      return(true);
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

All we do here is send a command to our `DownloadCheckService` to see if a download is available.

Step #3: Defining Our Event

Our `IntentService` will do the work of updating the book in the background. However, we will want to let the rest of the app know when the book is updated. In particular, the `ModelFragment`, if it exists, needs to know that there is a new set of book contents to display. To accomplish this, we can use another event on our `EventBus`, a `BookUpdatedEvent` in this case.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookUpdatedEvent` as the name and click OK to create the empty class.

Step #4: Defining Our JSON

Under the covers, Retrofit uses GSON for parsing the JSON it retrieves from the Web service (or other URL). Hence, just as we needed to define a Java class that models

our JSON for the book contents, we need a Java class that models the data we will get from our server as to whether or not a book update is available.

That JSON looks like:

```
{
  "updatedOn": "20120512",
  "updateUrl": "http://misc.commonsware.com/WarOfTheWorlds-Update.zip"
}
```

We can create a `BookUpdateInfo` class that mimics this structure.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookUpdateInfo` as the name and click OK to create the empty class.

Then, with `BookUpdateInfo` open in the editor, paste in the following class definition:

```
package com.commonsware.empublite;

public class BookUpdateInfo {
  String updatedOn;
  String updateUrl;
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/BookUpdateInfo.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Step #5: Defining Our Retrofit Interface

Retrofit then needs a Java *interface* that provides most of the details for how to fetch our JSON and convert it into a Java object. In our case, we will be using an HTTP `GET` operation to retrieve the JSON, and so we will use the Retrofit `@GET` annotation to point to a path on a server pointing to that JSON.

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookUpdateInterface` as the name, switch the "Kind" to be "Interface", and click OK to create the empty interface.

Then, with `BookUpdateInterface` open in the editor, paste in the following interface definition:

```
package com.commonsware.empublite;

import retrofit2.Call;
import retrofit2.http.GET;

public interface BookUpdateInterface {
  @GET("/misc/empublite-update.json")
  Call<BookUpdateInfo> update();
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/BookUpdateInterface.java)

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Here, we define our interface as having an `update()` method, returning an instance of our `BookUpdateInfo` structure, with the `@GET` annotation pointing to a path where the corresponding JSON can be found on a server to be designated later.

Step #6: Retrieving Our JSON Via Retrofit

Now, we can actually use Retrofit to retrieve our `BookUpdateInfo` and see if we have a book update.

First, we need to add the `INTERNET` permission to our app, as we are going to be downloading materials from the `INTERNET`. Add the following `<uses-permission>` element as a child of the root `<manifest>` element in `AndroidManifest.xml`:

```
  <uses-permission android:name="android.permission.INTERNET" />
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/AndroidManifest.xml)

Next, in `DownloadCheckService`, add an `OUR_BOOK_DATE` static data member, representing the edit date of the book baked into our APK, in YYYYMMDD format:

```
  private static final String OUR_BOOK_DATE="20120418";
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

Then, add a `getUpdateUrl()` method to `DownloadCheckService`:

```
private String getUpdateUrl() throws IOException {
  Retrofit retrofit=
    new Retrofit.Builder()
      .baseUrl("https://commonsware.com")
      .addConverterFactory(GsonConverterFactory.create())
      .build();
  BookUpdateInterface updateInterface=
    retrofit.create(BookUpdateInterface.class);
  BookUpdateInfo info=updateInterface.update().execute().body();

  if (info.updatedOn.compareTo(OUR_BOOK_DATE) > 0) {
    return(info.updateUrl);
  }

  return(null);
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

Here, we create a `Retrofit` instance, pointing to the server that is our "Web service" (really a static JSON file, but that does not matter from the standpoint of the client code). We then use the `Retrofit` instance to create an instance of a `BookUpdateInterface` implementation, code-generated by Retrofit. We then call `update()` on that object to get our `BookUpdateInfo`. If the date in the `updatedOn` field of our `BookUpdateInfo` is newer than `OUR_BOOK_DATE`, we return the `updateUrl` field of the `BookUpdateInfo`, which will be a URL pointing to a ZIP archive containing the updated book. If the `updatedOn` value is older than `OUR_BOOK_DATE`, we return `null` to signify that no updates are available.

This is not a particularly well-optimized approach. In particular, we never take into account that, once we have downloaded an update, we are only interested in updates newer than the one we downloaded. As it stands, we always compare the `updatedOn` value to `OUR_BOOK_DATE`, not the last `updatedOn` value that we used. A production-grade app would aim to handle this, such as by saving the last-used `updatedOn` value in a `SharedPreferences` and comparing against it, where available.

Finally, update `onHandleIntent()` to call `getUpdateUrl()`:

```
@Override
protected void onHandleIntent(Intent intent) {
  try {
    String url=getUpdateUrl();

    if (url != null) {
```

```
      // do something really cool here
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
          "Exception downloading update", e);
  }
}
```

Step #7: Downloading the Update

While the above code gets us the URL of the ZIP archive, it does not actually download it. We need more code to accomplish that.

Add a `private static final String` data member named `UPDATE_FILENAME` to `DownloadCheckService`, representing the name of the file for the downloaded ZIP file:

```
private static final String UPDATE_FILENAME="book.zip";
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

Then, in `DownloadCheckService`, add the following `download()` method:

```
private File download(String url) throws IOException {
  File output=new File(getFilesDir(), UPDATE_FILENAME);

  if (output.exists()) {
    output.delete();
  }

  OkHttpClient client=new OkHttpClient();
  Request request=new Request.Builder().url(url).build();
  Response response=client.newCall(request).execute();
  BufferedSink sink=Okio.buffer(Okio.sink(output));

  sink.writeAll(response.body().source());
  sink.close();

  return(output);
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

This method deletes the existing output file if it exists, then uses OkHttp (and its Okio transitive dependency) to download the book, writing the results to the designated output file.

Then, update `onHandleIntent()` in `DownloadCheckService` to call `download()` when we have something to download:

```
@Override
protected void onHandleIntent(Intent intent) {
  try {
    String url=getUpdateUrl();

    if (url != null) {
      File book=download(url);

      // do something almost as cool here

      book.delete();
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
          "Exception downloading update", e);
  }
}
```

Here, we delete the file after downloading it, so we do not clutter up our internal storage with the downloaded ZIP. This would appear to defeat the purpose of downloading the ZIP file in the first place, but we will add some code to use the ZIP file in the next step of the tutorial.

Step #8: Unpacking the Update

The last step in the book-download process is to unpack the ZIP archive onto internal storage, so we can start using the downloaded contents.

Add a `static final String` data member named `UPDATE_BASEDIR` to `DownloadCheckService`:

```
static final String UPDATE_BASEDIR="updates";
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

This will point to the directory on internal storage where the latest book update will reside.

Then, update `onHandleIntent()` on `DownloadCheckService` once again, this time to add in a call to `ZipUtils.unzip()` and some other necessary changes:

```
@Override
protected void onHandleIntent(Intent intent) {
  try {
    String url=getUpdateUrl();

    if (url != null) {
      File book=download(url);
      File updateDir=new File(getFilesDir(), UPDATE_BASEDIR);

      updateDir.mkdirs();
      ZipUtils.unzip(book, updateDir);
      book.delete();
      EventBus.getDefault().post(new BookUpdatedEvent());
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
      "Exception downloading update", e);
  }
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

Here, we:

- Create the `UPDATE_BASEDIR` directory if it does not already exist
- Call `ZipUtils.unzip()` to unZIP the ZIP file into that directory
- Post a `BookUpdatedEvent` to signify that a book update is ready

`ZipUtils` is a class from the CWAC-Security library that we added to our project back in Tutorial #6. Its `unzip()` method handles a variety of possible flaws in the ZIP archive that might be injected by an attacker who is intercepting our communications with the book update server.

Despite that, this update logic is a bit sloppy. It is possible that different book updates will have different files, and our `UPDATE_BASEDIR` will have some extra files as a result. Ideally, we should clean out `UPDATE_BASEDIR` before unpacking the ZIP

archive. Adding in some recursive delete-all-the-files-in-a-directory logic is left as an exercise for the reader.

At this point, `DownloadCheckService` should resemble:

```
package com.commonsware.empublite;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.security.ZipUtils;
import org.greenrobot.eventbus.EventBus;
import java.io.File;
import java.io.IOException;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;

public class DownloadCheckService extends IntentService {
  private static final String OUR_BOOK_DATE="20120418";
  private static final String UPDATE_FILENAME="book.zip";
  static final String UPDATE_BASEDIR="updates";

  public DownloadCheckService() {
    super("DownloadCheckService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    try {
      String url=getUpdateUrl();

      if (url != null) {
        File book=download(url);
        File updateDir=new File(getFilesDir(), UPDATE_BASEDIR);

        updateDir.mkdirs();
        ZipUtils.unzip(book, updateDir);
        book.delete();
        EventBus.getDefault().post(new BookUpdatedEvent());
      }
    }
    catch (Exception e) {
```

```
      Log.e(getClass().getSimpleName(),
        "Exception downloading update", e);
    }
  }

  private String getUpdateUrl() throws IOException {
    Retrofit retrofit=
      new Retrofit.Builder()
        .baseUrl("https://commonsware.com")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
    BookUpdateInterface updateInterface=
      retrofit.create(BookUpdateInterface.class);
    BookUpdateInfo info=updateInterface.update().execute().body();

    if (info.updatedOn.compareTo(OUR_BOOK_DATE) > 0) {
      return(info.updateUrl);
    }

    return(null);
  }

  private File download(String url) throws IOException {
    File output=new File(getFilesDir(), UPDATE_FILENAME);

    if (output.exists()) {
      output.delete();
    }

    OkHttpClient client=new OkHttpClient();
    Request request=new Request.Builder().url(url).build();
    Response response=client.newCall(request).execute();
    BufferedSink sink=Okio.buffer(Okio.sink(output));

    sink.writeAll(response.body().source());
    sink.close();

    return(output);
  }
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/DownloadCheckService.java)

Step #9: Using the Update

All this work is nice. However, nothing else in the app knows about this `UPDATE_BASEDIR` copy of the book to actually display it.

In fact, we have two scenarios to consider:

- The user taps the update action bar item, and we download the update and want to show the updated book to the user right now
- Later on, when the user opens the book, we need to realize that we already have an update and use it, rather than using the copy baked into the APK

That will require some changes to our data model, how we populate it from `ModelFragment`, and how we use the results in our `ContentsAdapter`.

First, add a `File baseDir` data member to `BookContents`, along with an accompanying setter method:

```
File baseDir=null;

void setBaseDir(File baseDir) {
  this.baseDir=baseDir;
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/BookContents.java)

Then, add a `getChapterPath()` method to `BookContents` that uses `getChapterFile()` for getting the relative path from the book's JSON, then uses that in conjunction with `baseDir` or the `android_asset` path to come up with a full `WebView`-friendly path to the file, whether it is in assets or a local file:

```
String getChapterPath(int position) {
  String file=getChapterFile(position);

  if (baseDir==null) {
    return("file:///android_asset/book/" + file);
  }

  return(Uri.fromFile(new File(baseDir, file)).toString());
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/BookContents.java)

Next, change the `getItem()` method on `ContentsAdapter` to use this new `getChapterPath()` method on `BookContents`:

```
@Override
public Fragment getItem(int position) {
  return(SimpleContentFragment.newInstance(contents.getChapterPath(position)));
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/ContentsAdapter.java)

Then, modify the `run()` method of the `LoadThread` in `ModelFragment` to try to use the update:

```
    @Override
    public void run() {
      prefs.set(PreferenceManager.getDefaultSharedPreferences(ctxt));

      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
      Gson gson=new Gson();
      File baseDir=
        new File(ctxt.getFilesDir(),
          DownloadCheckService.UPDATE_BASEDIR);

      try {
        InputStream is;

        if (baseDir.exists()) {
          is=new FileInputStream(new File(baseDir, "contents.json"));
        }
        else {
          is=ctxt.getAssets().open("book/contents.json");
        }

        BufferedReader reader=
          new BufferedReader(new InputStreamReader(is));

        contents.set(gson.fromJson(reader, BookContents.class));

        if (baseDir.exists()) {
          contents.get().setBaseDir(baseDir);
        }

        EventBus.getDefault().post(new BookLoadedEvent(getBook()));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
```

```
    }
  }
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

Here, we do the following, in addition to our original logic:

- See if the `UPDATE_BASEDIR` directory exists or not
- If it does, we use the `contents.json` in it; otherwise, we fall back to the one in `assets/` as before
- Update the `BookContents` with the update directory if we used that for loading the contents

This will handle the case where an update exists when we fire up the app and go to view the book. However, we still need some code that responds to the `BookUpdatedEvent` and arranges to use the updated contents at that point.

With that in mind, augment `onAttach()` on `ModelFragment` to register with the `EventBus`:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);

  EventBus.getDefault().register(this);

  if (contents.get()==null) {
    new LoadThread(host).start();
  }
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

We also now need a corresponding `onDetach()` method on `ModelFragment` to unregister from the `EventBus`:

```
@Override
public void onDetach() {
  EventBus.getDefault().unregister(this);

  super.onDetach();
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

Finally, we can respond to the `BookUpdatedEvent`, via a new `onEventBackgroundThread()` method on `ModelFragment`:

```
@SuppressWarnings("unused")
@Subscribe(threadMode =ThreadMode.BACKGROUND)
public void onBookUpdated(BookUpdatedEvent event) {
  if (getActivity()!=null) {
    new LoadThread(getActivity()).start();
  }
}
```

(from EmPubLite-AndroidStudio/T16-Update/EmPubLite/app/src/main/java/com/commonsware/empublite/ModelFragment.java)

The name `threadMode =ThreadMode.BACKGROUND` signals to the `EventBus` that we want to receive this event on a background thread. In our case, the event is posted on a background thread (the one from the `IntentService`). Hence, our `onBookUpdated()` method is called on that thread. If, however, we were to post a `BookUpdatedEvent` from the main application thread, `EventBus` would deliver our `BookUpdatedEvent` to `onBookUpdated()` on an `EventBus`-supplied background thread, to ensure that we do not tie up the main application thread.

Here, we just kick off a fresh `LoadThread` to reload the `BookContents`, assuming that the user has not just pressed BACK or otherwise destroyed our activity. The new `LoadThread` will see that the update is available and use it, posting its own event to have our UI layer apply the update to the screen.

At this point, if you build and run the app, you will see the update action bar item:

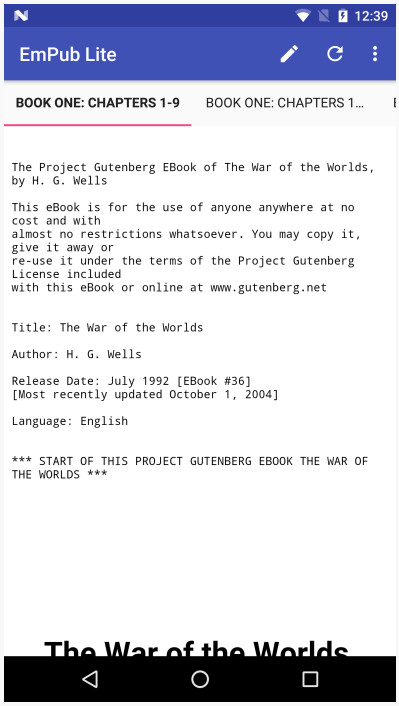


Figure 310: The New Action Bar Item

Swiping back to the first page in the `ViewPager`, tapping that action bar item, and waiting a few moments, should cause your book to be updated with new contents downloaded from the Internet:

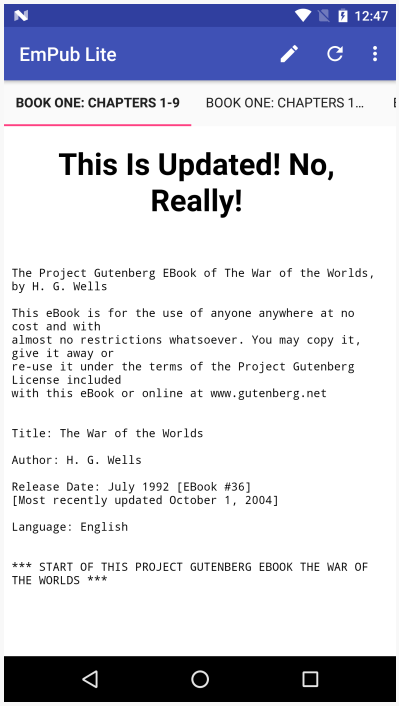


Figure 311: The Updated Content

In Our Next Episode…

… we will move some fragments into a sidebar on large-screen devices, like tablets.

Large-Screen Strategies and Tactics

So far, we have been generally ignoring screen size. With the vast majority of Android devices being in a fairly narrow range of sizes (3" to just under 5"), ignoring size while learning is not a bad approach. However, when it comes time to create a production app, you are going to want to strongly consider how you are going to handle other sizes, mostly larger ones (e.g., tablets).

Objective: Maximum Gain, Minimum Pain

What you want is to be able to provide a high-quality user experience without breaking your development budget — time *and* money — in the process.

An app designed around a phone, by default, may look fairly lousy on a tablet. That is because Android is simply going to try to stretch your layouts and such to fill the available space. While that will work, technically, the results may be unpleasant, or at least ineffective. If we have the additional room, it would be nice to allow the user to do something with that room.

At the same time, though, you do not have an infinite amount of time to be dealing with all of this. After all, there are a variety of tablet sizes. While ~7" and ~10" screens are the most common, there are certainly others that are reasonably popular (e.g., Amazon's Kindle Fire HD 8.9").

The Fragment Strategy

Some apps will use the additional space of a large screen directly. For example, a painting app would use that space mostly to provide a larger drawing canvas upon which the user can attempt to become the next Rembrandt, Picasso, or Pollock. The

app might elect to make more tools available directly on the screen as well, versus requiring some sort of pop-up to appear to allow the user to change brush styles, choose a different color, and so forth.

However, this can be a lot of work.

Some apps can make a simplifying assumption: the tablet UI is really a bunch of phone-sized layouts, stitched together. For example, if you take a 10" tablet in landscape, it is about the same size as two or three phones side-by-side. Hence, one could imagine taking the smarts out of a few activities and having them be adjacent to one another on a tablet, versus having to be visible only one at a time as they are on phones.

For example, consider the original edition of the Gmail app for Android.

On a phone, you would see conversations in a particular label on one screen:

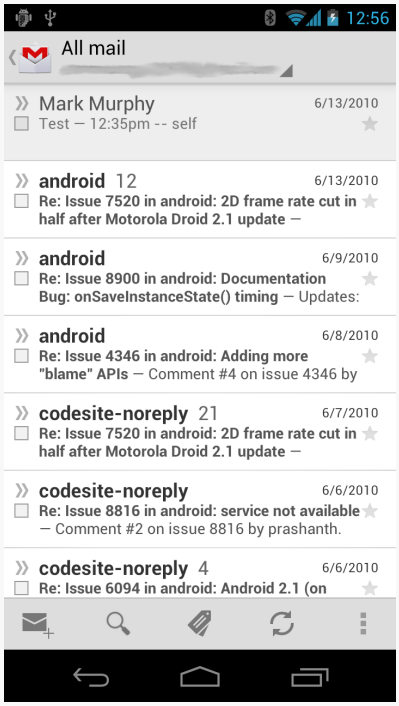


Figure 312: Gmail, On a Galaxy Nexus, Showing Conversations

... and the list of labels on another screen:

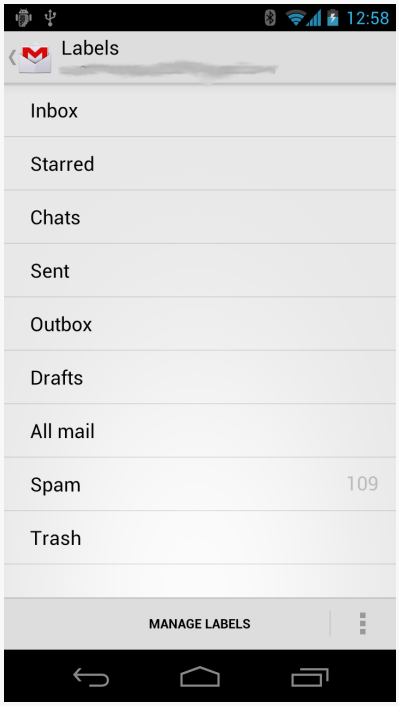


Figure 313: Gmail, On a Galaxy Nexus, Showing Labels

... and the list of messages in some selected conversation in a third screen:

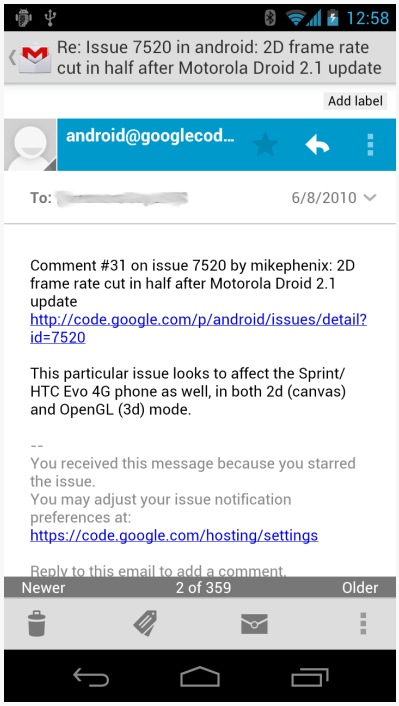


Figure 314: Gmail, On a Galaxy Nexus, Showing Messages

Whereas on a 7" tablet, you would see the list of labels and the conversations in a selected label at the same time:

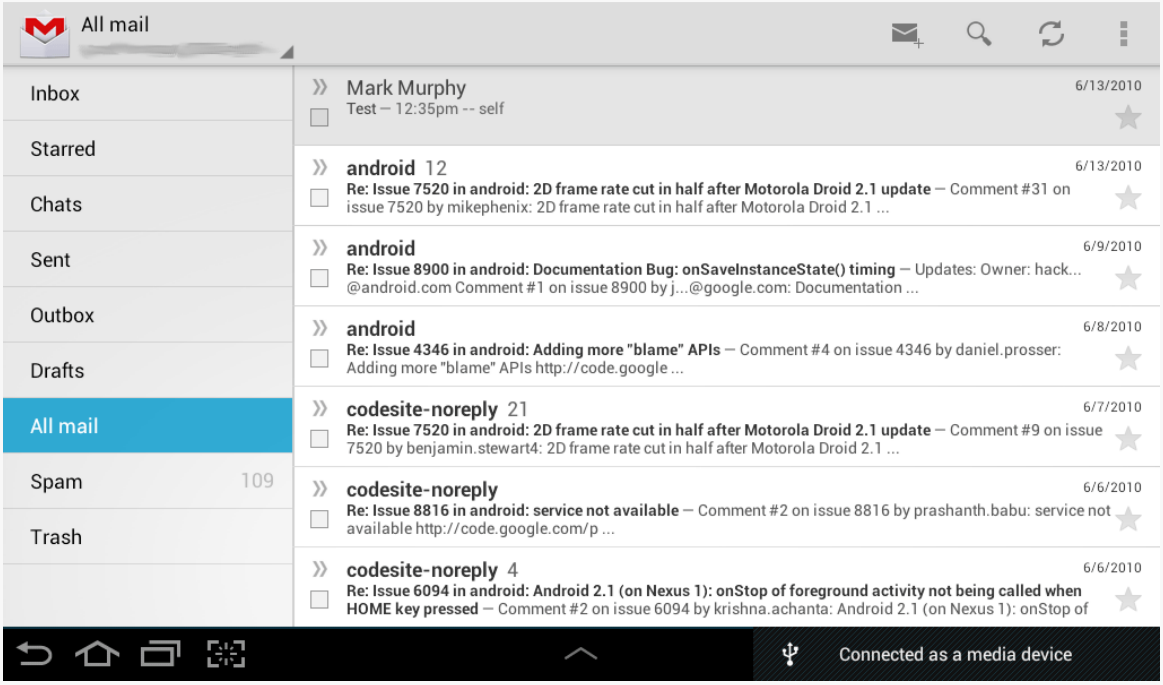


Figure 315: Gmail, On a Galaxy Tab 2, Showing Labels and Conversations

On that 7" tablet, tapping on a specific conversation brings up the list of messages for that conversation in a new screen. But, on a 10" tablet, tapping on a specific conversation showed that conversation, plus the list of conversations, side-by-side:

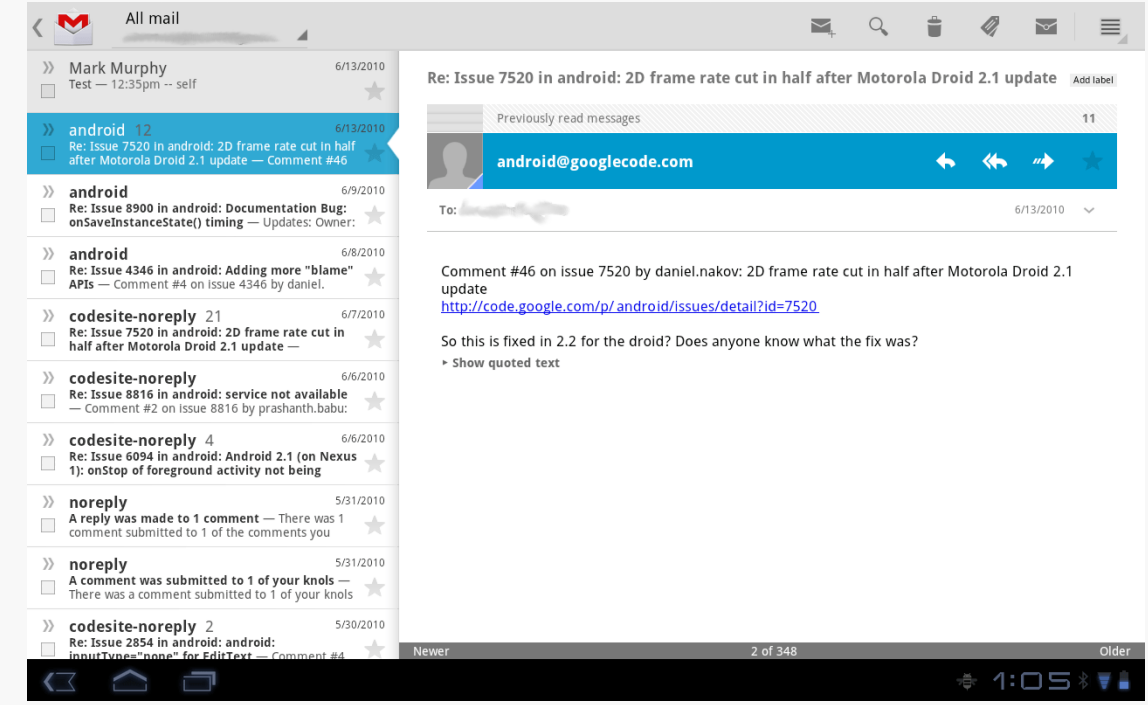


Figure 316: Gmail, On a Motorola XOOM, Showing Conversations and Messages

Yet all of that was done with one app with very little redundant logic, by means of fragments.

The list-of-labels, list-of-conversations, and list-of-messages bits of the UI were implemented as fragments. On a smaller screen (e.g., a phone), each one is displayed by an individual activity. Yet, on a larger screen (e.g., a tablet), more than one fragment is displayed by a single activity. In fact — though it will not be apparent from the static screenshots — on the 10" tablet, the activity showed *all three fragments*, using animated effects to slide the list of labels off-screen and the list of conversations over to the left slot when the user taps on a conversation to show the messages.

The vision, therefore, is to organize your UI into fragments, then choose which fragments to show in which circumstances based on available screen space:

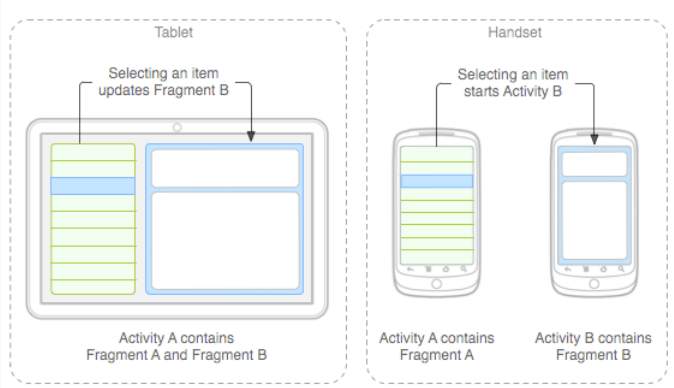


Figure 317: Tablets vs. Handsets (image courtesy of Android Open Source Project)

Changing Layout

One solution is to say that you have the same fragments for all devices and all configurations, but that the sizing and positioning of those fragments varies. This is accomplished by using different layouts for the activity, ones that provide the sizing and positioning rules for the fragments.

So far, most of our fragment examples have been focused on activities with a single fragment, like you might use on smaller screens (e.g., phones). However, activities can most certainly have more than one fragment, though you will need to provide the "slots" into which to plug those fragments.

For example, you could have the following in `res/layout-w720dp/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
```

```
    android:layout_height="match_parent"
  />
</LinearLayout>
```

(from LargeScreen/EU4You/app/src/main/res/layout-w720dp/main.xml)

Here we have a horizontal `LinearLayout` holding a pair of `FrameLayout` containers. Each of those `FrameLayout` containers will be a slot to load in a fragment, using code like:

```
getSupportFragmentManager().beginTransaction()
                                .add(R.id.countries, someFragmentHere)
                                .commit();
```

In principle, you could also have a `res/layout-h720dp/main.xml` that holds both of the same `FrameLayout` containers, but just in a `vertical` `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_height="0dp"
    android:layout_width="match_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_height="0dp"
    android:layout_width="match_parent"
  />
</LinearLayout>
```

As the user rotates the device, the fragments will go in their appropriate slots.

Changing Fragment Mix

However, for larger changes in screen size, you will probably need to have larger changes in your fragments. The most common pattern is to have fewer fragments on-screen for an activity on a smaller-screen device (e.g., one fragment at a time on

a phone) and more fragments on-screen for an activity on a larger-screen device (e.g., two fragments at a time on a tablet).

So, for example, as the counterpart to the `res/layout-w720dp/main.xml` shown in the previous section, you might have a `res/layout/main.xml` that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

(from LargeScreen/EU4You/app/src/main/res/layout/main.xml)

This provides a single slot, `R.id.countries`, for a fragment, one that fills the screen. For a larger-screen device, held in landscape, you would use the two-fragment layout; for anything else (e.g., tablet in portrait, or phone in any orientation), you would use the one-fragment layout.

Of course, the content that belongs in the second fragment would have to show up *somewhere*.

Sometimes, when you add another fragment for a large screen, you only want it to be there some of the time. For example, a digital book reader (like the one we are building in the tutorials) might normally take up the full screen with the reading fragment, but might display a sidebar fragment based upon an action bar item click or the like. If you would like the BACK button to reverse your `FragmentTransaction` that added the second fragment — so pressing BACK removes that fragment and returns you to the single-fragment setup — you can add `addToBackStack()` as part of your `FragmentTransaction` construction:

```
getSupportFragmentManager().beginTransaction()
                           .addToBackStack(null)
                           .replace(R.id.sidebar, f)
                           .commit();
```

We will see this in the next tutorial.

The Role of the Activity

So, what is the activity doing?

First, the activity is the one loading the overall layout, the one indicating which fragments should be loaded (e.g., the samples shown above). The activity is responsible for populating those "slots" with the appropriate fragments. It can determine which fragments to create based on which slots exist, so it would only try to create a fragment to go in `R.id.details` if there actually is an `R.id.details` slot to use.

Next, the activity is responsible for handling any events that are triggered by UI work in a fragment (e.g., user clicking on a `ListView` item), whose results should impact other fragments (e.g., displaying details of the clicked-upon `ListView` item). The activity knows which fragments exist at the present time. So, the activity can either call some method on the second fragment if it exists, or it can call `startActivity()` to pass control to another activity that will be responsible for the second fragment if it does not exist in the current activity.

Finally, the activity is generally responsible for any model data that spans multiple fragments. Whether that model data is held in a "model fragment" (as outlined in the chapter on fragments) or somewhere else is up to you.

Fragment Example: The List-and-Detail Pattern

This will make a bit more sense as we work through another example, this time focused on a common pattern: a list of something, where clicking on the list brings up details on the item that was clicked upon. On a larger-screen device, in landscape, both pieces are typically displayed at the same time, side-by-side. On smaller-screen devices, and sometimes even on larger-screen devices in portrait, only the list is initially visible — tapping on a list item brings up some other activity to display the details.

Describing the App

The sample app for this section is `LargeScreen/EU4You`. This app has a list of member nations of the European Union (EU). Tapping on a member nation will display the mobile Wikipedia page for that nation in a `WebView` widget.

The data model — such as it is and what there is of it — consists of a `Country` class which holds onto the country name (as a string resource ID), flag (as a drawable resource ID), and mobile Wikipedia URL (as another string resource ID):

```
Country(int name, int flag, int url) {
  this.name=name;
  this.flag=flag;
  this.url=url;
}
```

The `Country` class has a static `ArrayList` of `Country` objects representing the whole of the EU, initialized in a static initialization block:

```
  static {
    EU.add(new Country(R.string.austria, R.drawable.austria,
                       R.string.austria_url));
    EU.add(new Country(R.string.belgium, R.drawable.belgium,
                       R.string.belgium_url));
    EU.add(new Country(R.string.bulgaria, R.drawable.bulgaria,
                       R.string.bulgaria_url));
    EU.add(new Country(R.string.cyprus, R.drawable.cyprus,
                       R.string.cyprus_url));
    EU.add(new Country(R.string.czech_republic,
                       R.drawable.czech_republic,
                       R.string.czech_republic_url));
    EU.add(new Country(R.string.denmark, R.drawable.denmark,
                       R.string.denmark_url));
    EU.add(new Country(R.string.estonia, R.drawable.estonia,
                       R.string.estonia_url));
    EU.add(new Country(R.string.finland, R.drawable.finland,
                       R.string.finland_url));
    EU.add(new Country(R.string.france, R.drawable.france,
                       R.string.france_url));
    EU.add(new Country(R.string.germany, R.drawable.germany,
                       R.string.germany_url));
    EU.add(new Country(R.string.greece, R.drawable.greece,
                       R.string.greece_url));
    EU.add(new Country(R.string.hungary, R.drawable.hungary,
                       R.string.hungary_url));
    EU.add(new Country(R.string.ireland, R.drawable.ireland,
                       R.string.ireland_url));
    EU.add(new Country(R.string.italy, R.drawable.italy,
                       R.string.italy_url));
    EU.add(new Country(R.string.latvia, R.drawable.latvia,
                       R.string.latvia_url));
    EU.add(new Country(R.string.lithuania, R.drawable.lithuania,
                       R.string.lithuania_url));
    EU.add(new Country(R.string.luxembourg, R.drawable.luxembourg,
                       R.string.luxembourg_url));
    EU.add(new Country(R.string.malta, R.drawable.malta,
                       R.string.malta_url));
```

```
    EU.add(new Country(R.string.netherlands, R.drawable.netherlands,
                       R.string.netherlands_url));
    EU.add(new Country(R.string.poland, R.drawable.poland,
                       R.string.poland_url));
    EU.add(new Country(R.string.portugal, R.drawable.portugal,
                       R.string.portugal_url));
    EU.add(new Country(R.string.romania, R.drawable.romania,
                       R.string.romania_url));
    EU.add(new Country(R.string.slovakia, R.drawable.slovakia,
                       R.string.slovakia_url));
    EU.add(new Country(R.string.slovenia, R.drawable.slovenia,
                       R.string.slovenia_url));
    EU.add(new Country(R.string.spain, R.drawable.spain,
                       R.string.spain_url));
    EU.add(new Country(R.string.sweden, R.drawable.sweden,
                       R.string.sweden_url));
    EU.add(new Country(R.string.united_kingdom,
                       R.drawable.united_kingdom,
                       R.string.united_kingdom_url));
  }
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/Country.java)

CountriesFragment

The fragment responsible for rendering the list of EU nations is `CountriesFragment`. It is a `ListFragment`, using a `CountryAdapter` to populate the list:

```
class CountryAdapter extends ArrayAdapter<Country> {
  CountryAdapter() {
    super(getActivity(), R.layout.row, R.id.name, Country.EU);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    CountryViewHolder wrapper=null;

    if (convertView == null) {
      convertView=
          LayoutInflater.from(getActivity()).inflate(R.layout.row,
                                                     parent, false);
      wrapper=new CountryViewHolder(convertView);
      convertView.setTag(wrapper);
    }
    else {
      wrapper=(CountryViewHolder)convertView.getTag();
    }
```

```
      wrapper.populateFrom(getItem(position));

      return(convertView);
    }
  }
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/CountriesFragment.java)

This adapter is somewhat more complex than the ones we showed in the chapter on selection widgets. We will get into what `CountryAdapter` is doing, and the `CountryViewHolder` it references, in a later chapter of this book. Suffice it to say for now that the rows in the list contain both the country name and its flag.

When the user taps on a row in our `ListView`, something needs to happen – specifically, the details of that country need to be displayed. However, displaying those details is not the responsibility of `CountriesFragment`, as it simply displays the list of countries and nothing else. Hence, we need to pass the event up to the hosting activity to handle.

We could accomplish this using an event bus, as seen in other examples earlier in the book. The `EU4You` series of samples, though, use a different approach, referred to as the contract pattern. In this pattern, the fragment defines an interface, which is the "contract" that all hosting activities of that fragment must implement. This requirement is enforced by the superclass, `ContractListFragment`:

```
/***
  Copyright (c) 2013 Jake Wharton
  Portions Copyright (c) 2013 CommonsWare, LLC

  Licensed under the Apache License, Version 2.0 (the "License"); you may not
  use this file except in compliance with the License. You may obtain a copy
  of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required
  by applicable law or agreed to in writing, software distributed under the
  License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
  OF ANY KIND, either express or implied. See the License for the specific
  language governing permissions and limitations under the License.

  Covered in detail in the book _The Busy Coder's Guide to Android Development_
    https://commonsware.com/Android
 */

// derived from https://gist.github.com/JakeWharton/2621173

package com.commonsware.android.eu4you;
```

```
import android.app.Activity;
import android.app.ListFragment;

public class ContractListFragment<T> extends ListFragment {
  private T contract;

  @SuppressWarnings("unchecked")
  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);

    try {
      contract=(T)activity;
    }
    catch (ClassCastException e) {
      throw new IllegalStateException(activity.getClass()
                                              .getSimpleName()
          + " does not implement contract interface for "
          + getClass().getSimpleName(), e);
    }
  }

  @Override
  public void onDetach() {
    super.onDetach();

    contract=null;
  }

  public final T getContract() {
    return(contract);
  }
}
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/ContractListFragment.java)

`onAttach()` is called when the fragment has been attached to an activity, whether that is from when the activity was initially created, after a configuration change, or whenever. In those cases, we cast the activity to be the contract interface (provided via the data type in the declaration), raising an exception if the cast fails. Subclasses can then access the contract object via the `getContract()` method.

`CountriesFragment` inherits from `ContractListFragment` and defines its contract. Hence, any activity that hosts a `CountriesFragment` is responsible for implementing

this contract interface, so we can call `onCountrySelected()` when the user clicks on a row in the list:

```
  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    if (getContract().isPersistentSelection()) {
      getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
      l.setItemChecked(position, true);
    }
    else {
      getListView().setChoiceMode(ListView.CHOICE_MODE_NONE);
    }

    getContract().onCountrySelected(Country.EU.get(position));
  }
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/CountriesFragment.java)

`CountriesFragment` also has quite a bit of code dealing with clicked-upon rows being in an "activated" state. This provides visual context to the user and is often used in the list-and-details pattern. For example, in the tablet renditions of Gmail shown earlier in this chapter, you will notice that the list on the left (e.g., list of labels) has one row highlighted with a blue background. This is the "activated" row, and it indicates the context for the material in the adjacent fragment (e.g., list of conversations in the label). Managing this "activated" state is a bit beyond the scope of this section, however, so we will delay discussion of that topic to a later chapter in this book.

DetailsFragment

The details to be displayed come in the form of a URL to a mobile Wikipedia page for a country, designed to be displayed in a `WebView`. The EU4You sample app makes use of the same `WebViewFragment` that we saw earlier in this book, such as in the tutorials. `DetailsFragment` itself, therefore, simply needs to expose some method to allow a hosting activity to tell it what URL to display:

```
package com.commonsware.android.eu4you;

import android.os.Bundle;
import android.view.View;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.webkit.WebViewFragment;
```

```
public class DetailsFragment extends WebViewFragment {
  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    getWebView().setWebViewClient(new URLHandler());
  }

  public void loadUrl(String url) {
    getWebView().loadUrl(url);
  }

  private static class URLHandler extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
      view.loadUrl(url);

      return(true);
    }
  }
}
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/DetailsFragment.java)

You will notice that this fragment is not retained via `setRetainInstance()`. That is because, as you will see, we will not always be displaying this fragment. Fragments that are displayed in some configurations (e.g., landscape) but not in others (e.g., portrait), where a device might change between those configurations at runtime, cannot be retained without causing crashes.

You will also notice that this fragment uses `setWebViewClient()` to associate a `URLHandler` with the `WebView`. This `URLHandler` class simply forces all URLs back into the `WebView`, as opposed to launching a browser. Wikipedia now uses HTTPS for many pages, and it uses HTTP Strict Transport Security (HSTS) to redirect HTTP requests to their HTTPS counterparts as appropriate. The mobile Wikipedia URLs used in the app all have the `https` scheme, and so in theory there should be no server-side redirects. But, just in case, the `URLHandler` ensures that such redirects will stay within the `WebView`.

The Activities

Our launcher activity is also named `EU4You`. It uses two of the layouts shown above. Both are `main.xml`, but one is in `res/layout-w720dp/`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

(from LargeScreen/EU4You/app/src/main/res/layout-w720dp/main.xml)

The other is in `res/layout/`:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

(from LargeScreen/EU4You/app/src/main/res/layout/main.xml)

Both have a `FrameLayout` for the `CountriesFragment` (`R.id.countries`), but only the `res/layout-w720dp/` edition has a `FrameLayout` for the `DetailsFragment` (`R.id.details`).

Here is the complete implementation of the `EU4You` activity:

```
package com.commonsware.android.eu4you;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class EU4You extends Activity implements
    CountriesFragment.Contract {
  private CountriesFragment countries=null;
  private DetailsFragment details=null;
```

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    countries=
        (CountriesFragment)getFragmentManager().findFragmentById(R.id.countries);

    if (countries == null) {
      countries=new CountriesFragment();
      getFragmentManager().beginTransaction()
                                  .add(R.id.countries, countries)
                                  .commit();
    }

    details=
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);

    if (details == null && findViewById(R.id.details) != null) {
      details=new DetailsFragment();
      getFragmentManager().beginTransaction()
                                  .add(R.id.details, details).commit();
    }
  }

  @Override
  public void onCountrySelected(Country c) {
    String url=getString(c.url);

    if (details != null && details.isVisible()) {
      details.loadUrl(url);
    }
    else {
      Intent i=new Intent(this, DetailsActivity.class);

      i.putExtra(DetailsActivity.EXTRA_URL, url);
      startActivity(i);
    }
  }

  @Override
  public boolean isPersistentSelection() {
    return(details != null && details.isVisible());
  }
}
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/EU4You.java)

The job of `onCreate()` is to set up the UI. So, we:

- See if we already have an instance of `CountriesFragment`, by asking our `FragmentManager` to give us the fragment in the `R.id.countries` slot — this might occur if we underwent a configuration change, as `CountriesFragment` would be recreated in that case

- If we do not have a `CountriesFragment` instance, create one and execute a `FragmentTransaction` to load it into `R.id.countries` of our layout
- Find the `DetailsFragment` (which, since `DetailsFragment` is not retained, should always return `null`, but, as they say, "better safe than sorry")
- If we do not have a `DetailsFragment` *and* the layout has an `R.id.details` slot, create a `DetailsFragment` and execute the `FragmentTransaction` to put it in that slot... but otherwise do nothing

The net result is that `EU4You` can correctly handle either situation, where we have both fragments or just one.

Similarly, the `onCountrySelected()` method (required by the `Contract` interface) will see if we have our `DetailsFragment` or not (and whether it is visible, or is hidden because we created it but it is not visible in the current screen orientation). If we do, we just call `loadUrl()` on it, to populate the `WebView`. If we do not have a visible `DetailsFragment`, we need to do something to display one. In principle, we could elect to execute a `FragmentTransaction` to replace the `CountriesFragment` with the `DetailsFragment`, but this can get complicated. Here, we start up a separate `DetailsActivity`, passing the URL for the chosen `Country` in an `Intent` extra.

`DetailsActivity` is similar:

```
package com.commonsware.android.eu4you;

import android.app.Activity;
import android.os.Bundle;

public class DetailsActivity extends Activity {
  public static final String EXTRA_URL=
      "com.commonsware.android.eu4you.EXTRA_URL";
  private String url=null;
  private DetailsFragment details=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    details=(DetailsFragment)getFragmentManager()
                             .findFragmentById(android.R.id.content);

    if (details == null) {
      details=new DetailsFragment();

      getFragmentManager().beginTransaction()
```

```
                                .add(android.R.id.content, details)
                                .commit();
    }

    url=getIntent().getStringExtra(EXTRA_URL);
  }

  @Override
  public void onResume() {
    super.onResume();

    details.loadUrl(url);
  }
}
```

(from LargeScreen/EU4You/app/src/main/java/com/commonsware/android/eu4you/DetailsActivity.java)

We create the `DetailsFragment` and load it into the layout, capture the URL from the `Intent` extra, and call `loadUrl()` on the `DetailsFragment`. However, since we are executing a `FragmentTransaction`, the actual UI for the `DetailsFragment` is not created immediately, so we cannot call `loadUrl()` right away (otherwise, `DetailsFragment` will try to pass it to a non-existent `WebView`, and we crash). So, we delay calling `loadUrl()` to `onResume()`, at which point the `WebView` should exist.

The Results

On a larger-screen device, in landscape, we have both fragments, though there is nothing initially loaded into the `DetailsFragment`:

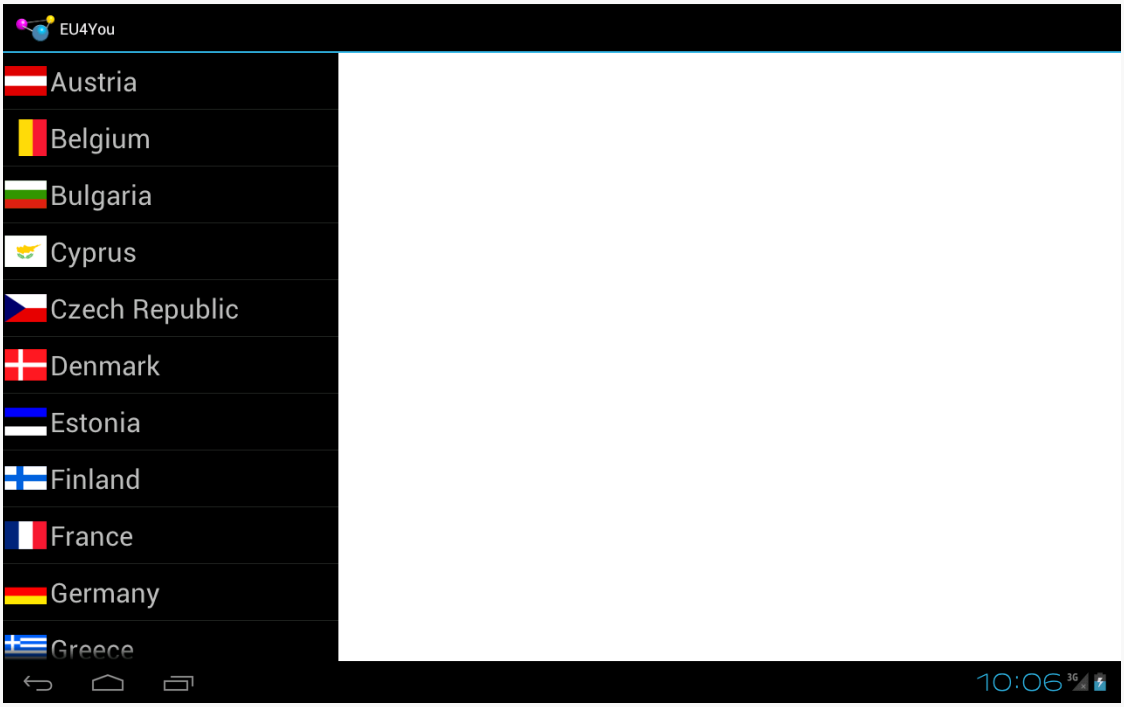


Figure 318: EU4You, On a Tablet Emulator, Landscape

Tapping on a country brings up the details on the right:

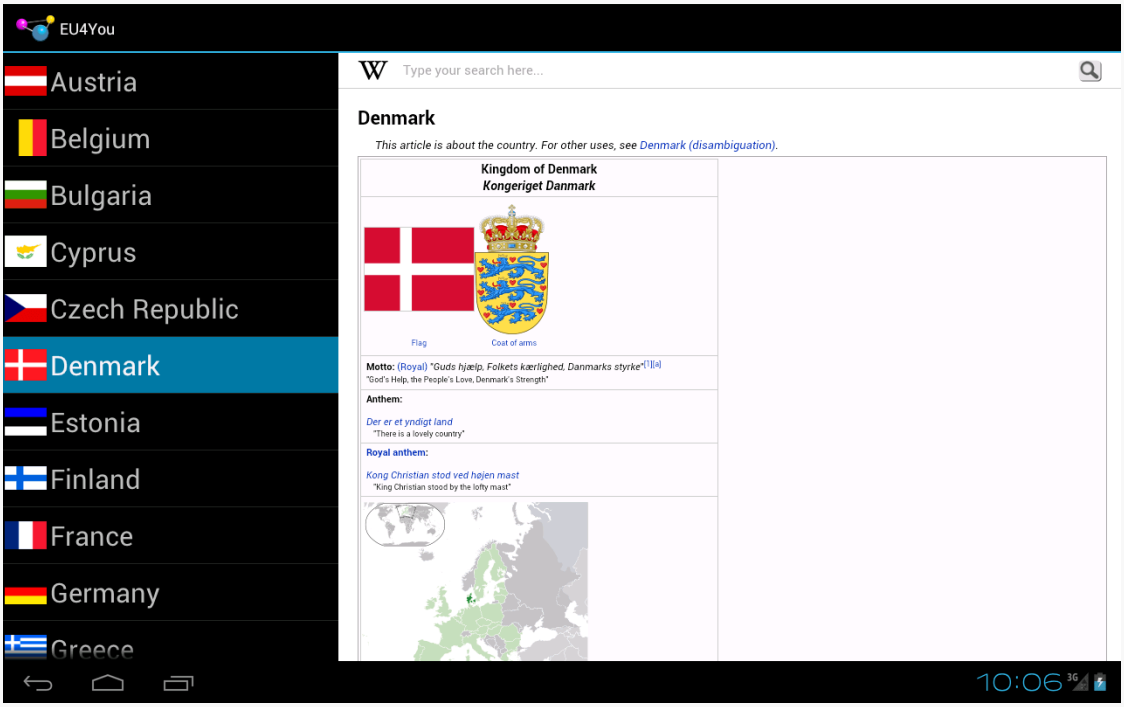


Figure 319: EU4You, On a Tablet Emulator, Landscape, With Details

In any other configuration, such as a smaller-screen device, we only see the `CountriesFragment` at the outset:

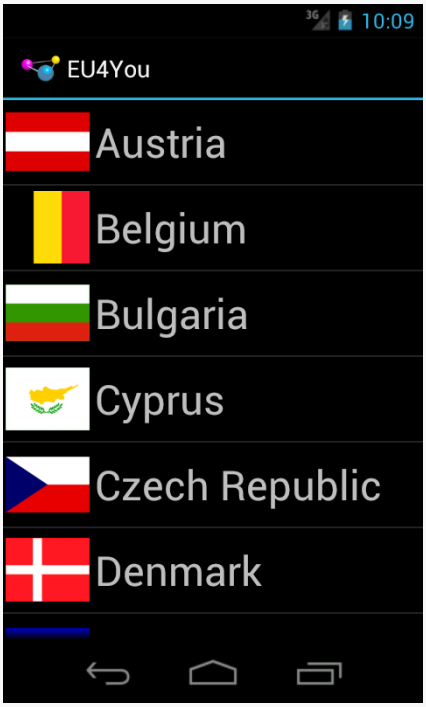


Figure 320: EU4You, On a Phone Emulator

Tapping on a country brings up the `DetailsFragment` full-screen in the `DetailsActivity`:

Figure 321: EU4You, On a Phone Emulator, Showing Details

Other Master-Detail Strategies

The EU4You sample from above is one way of approaching this master-detail pattern. It is not the only one. In this section, will we review other implementations of EU4You that use other techniques for implementating the master-detail pattern.

Static CountriesFragment

In the original `EU4You` activity, both fragments were dynamic, each added via a `FragmentTransaction`. `DetailsFragment` has to be dynamic, as whether or not it is visible depends upon screen size and orientation. However, there is no particular need for our `CountriesFragment` to be dynamic, as you will see in the `LargeScreen/EU4YouStaticCountries` sample project.

Here, our single-pane layout uses a `<fragment>` element to wire in the `CountriesFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  android:id="@+id/countries"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.eu4you3.CountriesFragment"
/>
```

(from LargeScreen/EU4YouStaticCountries/app/src/main/res/layout/main.xml)

Similarly, our dual-pane layout uses a `<fragment>` element for the `CountriesFragment`, alongside the `FrameLayout` for the details:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:baselineAligned="false"
  android:orientation="horizontal">

  <fragment
    android:id="@+id/countries"
    android:name="com.commonsware.android.eu4you3.CountriesFragment"
    android:layout_width="0px"
    android:layout_height="match_parent"
    android:layout_weight="30"/>

  <FrameLayout
    android:id="@+id/details"
    android:layout_width="0px"
    android:layout_height="match_parent"
    android:layout_weight="70"/>

</LinearLayout>
```

(from LargeScreen/EU4YouStaticCountries/app/src/main/res/layout-w720dp/main.xml)

Our `onCreate()` for `EU4You` is simpler, in that we do not need to mess with the `CountriesFragment` at all:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    details=
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);

    if (details == null && findViewById(R.id.details) != null) {
```

```
      details=new DetailsFragment();
      getFragmentManager().beginTransaction()
                                  .add(R.id.details, details).commit();
    }
  }
```

(from LargeScreen/EU4YouStaticCountries/app/src/main/java/com/commonsware/android/eu4you3/EU4You.java)

Neither `CountriesFragment` or anything involving the details necessarily needs to change.

Going With One Activity

You might wonder why we need to bother with `DetailsActivity`. After all, the `EU4You` activity is perfectly capable of showing the `DetailsFragment` in a second pane — why not have it display the `DetailsFragment` in the first pane as well, in single-pane scenarios? Surely, this will be much simpler, as we can dispense with the activity and its entry in the manifest!

Yes, this is possible. No, it is not simpler.

The reason for the complexity is now managing all of our possible mix of fragments. We already had to deal with the following possibilities:

- Single-pane, showing the countries
- Single-pane, showing the countries, but on a large screen in portrait mode, after the activity had been launched in landscape, so the `DetailsFragment` exists in the `FragmentManager`, but is not visible
- Dual-pane, showing both fragments

If we get rid of `DetailsActivity` and dump all the responsibility onto `EU4You`, we have more scenarios:

- Single-pane, showing the details, having replaced the countries via a `FragmentTransaction`
- Single-pane, showing the countries, after having shown the details and the user then pressing BACK

Basically, what we must do now is `replace()` the `CountriesFragment` with the `DetailsFragment`, when we are in single-pane mode, when the user taps on a country in the list. This requires a fairly extensive number of changes, as you will see in the `LargeScreen/EU4YouSingleActivity` sample project.

The Revised Layouts

In our single-pane mode, our one pane will either hold the `CountriesFragment` or the `DetailsFragment`, depending upon what the user has done. Right now, our `FrameLayout` is named `R.id.countries`, which was fine before, but now seems like an inappropriate name. So, the new project's layouts change this to `R.id.mainfrag`, without changing anything else:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/mainfrag"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

(from LargeScreen/EU4YouSingleActivity/app/src/main/res/layout/main.xml)

The New onCountrySelected()

The "simple" part of the changes comes in the revised `onCountrySelected()` method in EU4You:

```
  @Override
  public void onCountrySelected(Country c) {
    String url=getString(c.url);

    details.loadUrl(url);

    if (details.getId() != R.id.details) {
      getFragmentManager().beginTransaction()
                          .replace(R.id.mainfrag, details,
                                   TAG_DETAILS)
                          .addToBackStack(null).commit();
    }
  }
```

(from LargeScreen/EU4YouSingleActivity/app/src/main/java/com/commonsware/android/eu4you2/EU4You.java)

In our revised scenario, we will *always* have a `DetailsFragment`. The question is merely whether it is presently visible. Hence, we can call `loadUrl()` on `details` directly.

However, there are two possible scenarios for the status of our `DetailsFragment` at the point in time of `onCountrySelected()` being called:

1. It exists in the `details` `FrameLayout` of our dual-pane layout resource
2. It exists, perhaps due to a configuration change, but is not presently in a container

You might think that there would be a third scenario, where it is the visible fragment in the `mainfrag` `FrameLayout`. Indeed, sometimes `DetailsFragment` will be in that container... just not now. The only time that `onCountrySelected()` will be called is if the user tapped on an item in our `CountriesFragment`, which means that `CountriesFragment` must be in `mainfrag`.

The ID of a fragment, from `getId()`, is the ID of its container, when used with dynamic fragments. So, we check to see whether our `DetailsFragment` is in the `details` `FrameLayout` by comparing ID values. If they differ, then we `commit()` a `replace()` `FragmentTransaction` to put `DetailsFragment` into `mainfrag`. Note, though, that we use `addToBackStack()`, so if the user presses the BACK button, we will roll back this transaction and return to the `CountriesFragment`.

The New onCreate()

If you thought that was messy, you will not like the changes required to `onCreate()` of `EU4You` much more:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  countries=
      (CountriesFragment)getFragmentManager().findFragmentByTag(TAG_COUNTRIES);
  details=
      (DetailsFragment)getFragmentManager().findFragmentByTag(TAG_DETAILS);

  if (countries == null) {
    countries=new CountriesFragment();
    getFragmentManager().beginTransaction()
                                .add(R.id.mainfrag, countries,
                                     TAG_COUNTRIES).commit();
  }

  if (details == null) {
    details=new DetailsFragment();

    if (findViewById(R.id.details) != null) {
      getFragmentManager().beginTransaction()
                                .add(R.id.details, details,
                                     TAG_DETAILS).commit();
    }
  }
  else {
```

```
    if (details.getId() == R.id.mainfrag) {
      if (findViewById(R.id.details) != null) {
        getFragmentManager().popBackStackImmediate();
      }
    }
    else {
      getFragmentManager().beginTransaction().remove(details)
                          .commit();
    }

    if (findViewById(R.id.details) != null) {
      getFragmentManager().beginTransaction()
                          .add(R.id.details, details,
                               TAG_DETAILS).commit();
    }
  }
}
```

(from LargeScreen/EU4YouSingleActivity/app/src/main/java/com/commonsware/android/eu4you2/EU4You.java)

This sample is derived from the original EU4You sample, and so we are still using a `FragmentTransaction` to set up the `CountriesFragment` in `mainfrag`, if we did not create `CountriesFragment` earlier.

Dealing with `DetailsFragment`, though, is decidedly more complicated. The flow that we want is if we were in dual-pane mode and switch to single-pane mode, that we show the `CountriesFragment` in that single pane. If we switch from single-pane mode to dual-pane mode, both fragments will be shown, of course.

First, we have the case where our `DetailsFragment` does not yet exist. This is much like the original sample: we need to create the fragment and put it into the `details` `FrameLayout`, if the `details` `FrameLayout` exists.

If the `DetailsFragment` exists, we need to make sure that it winds up in the `details` `FrameLayout`, if one exists.

To do that, we first check its ID to see if it is presently located in `mainfrag`. If it is, and if we have a `details` `FrameLayout`, we have switched to dual-pane mode and need to pop our back stack, in preparation for moving the `DetailsFragment` to the `details` `FrameLayout`.

If the `DetailsFragment` exists but is not in `mainfrag`, we `remove()` it entirely.

Then, if the `DetailsFragment` exists, regardless of where it was before, we `add()` it to the `details` `FrameLayout`.

The "OMG! Our Fragments Have No Views!" Changes

In testing, there are now scenarios in which `CountriesFragment` is called with `onSaveInstanceState()`, but without its views having been created (i.e., `onCreateView()` was not called). This would cause us to fail when trying to use `getListView()`, as that method would return `null`, since the `ListView` did not exist. So, we modify `onSaveInstanceState()` to be a bit more robust:

```
  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    if (getView() != null) {
      state.putInt(STATE_CHECKED,
                   getListView().getCheckedItemPosition());
    }
  }
```

(from LargeScreen/EU4YouSingleActivity/app/src/main/java/com/commonsware/android/eu4you2/CountriesFragment.java)

We also need to beef up `DetailsFragment` a bit. Before, we relied on the fact that, on a configuration change, our extras on our `Intent` for `DetailsActivity` would still be available. Now, though, there is no `DetailsActivity`, which means that `DetailsFragment` has to maintain its state, so that we do not lose the URL we were viewing when the user rotates the screen or causes another configuration change. And, to top it off, we have the same potential issue as with `CountriesFragment`, where the fragment might exist but not have `onCreateView()` called (e.g., we were in dual-pane mode and switched to single-pane mode, and `DetailsFragment` has not yet been displayed), so we cannot assume that `getWebView()` will always return a non-`null` value.

To that end, `DetailsFragment` gets complicated:

```
package com.commonsware.android.eu4you2;

import android.os.Bundle;
import android.view.View;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.webkit.WebViewFragment;

public class DetailsFragment extends WebViewFragment {
  private static final String STATE_URL="url";
  private String url=null;
```

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  getWebView().setWebViewClient(new URLHandler());
}

@Override
public void onActivityCreated(Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);

  if (url == null && savedInstanceState != null) {
    url=savedInstanceState.getString(STATE_URL);
  }

  if (url != null) {
    loadUrl(url);
    url=null;
  }
}

@Override
public void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  if (url == null) {
    outState.putString(STATE_URL, getWebView().getUrl());
  }
  else {
    outState.putString(STATE_URL, url);
  }
}

void loadUrl(String url) {
  if (getView() == null) {
    this.url=url;
  }
  else {
    getWebView().loadUrl(url);
  }
}

private static class URLHandler extends WebViewClient {
  @Override
  public boolean shouldOverrideUrlLoading(WebView view, String url) {
    view.loadUrl(url);

    return(true);
```

```
    }
  }
}
```

(from LargeScreen/EU4YouSingleActivity/app/src/main/java/com/commonsware/android/eu4you2/DetailsFragment.java)

The `url` data member will temporarily hold the URL of the page we should be viewing, particularly when we have no `WebView` to work with. So, our `loadUrl()` method now puts the URL into `url` if we have no `WebView` or loads it into the `WebView` if the `WebView` exists. `onSaveInstanceState()` will put the URL — whether from `url` or from the `WebView` — into the state `Bundle`. `onActivityCreated()` will attempt to populate `url` from the `Bundle` (if we do not already have a URL), then use that to populate the `WebView` (which should exist if `onActivityCreated()` is called). `url` is set to `null` to indicate that the `WebView` holds our URL, once that is completed.

The Results

From a user experience standpoint, things have not significantly changed. The user still sees the list, still sees the details when tapping on an entry in the list, and still gets the dual-pane experience on larger screens.

However, the transition between the list and the details in single-pane mode is a bit faster, as a `FragmentTransaction` takes less time than does starting up another activity. However, by default, our `FragmentTransaction` does not apply any transition effects, and so the fragment changes "just happen" without any fades, zooms, or the like. It is certainly possible to specify fragment transition effects, if desired, though this is outside the scope of this chapter.

The Mashup Possibilities

It should be possible to combine the two revised versions of EU4You, having a single activity manage all the fragments, with `CountriesFragment` set up as a static fragment. The proof that this is possible is left to the reader.

The SlidingPaneLayout Variant

The R13 update to the Android Support package introduced `SlidingPaneLayout`, another way of handling this sort of master-detail pattern. `SlidingPaneLayout` significantly reduces the level of effort for setting up master-detail, as it handles all

of the "dirty work" of showing the different fragments in different scenarios (normal screen, large screen, etc.).

The Role of SlidingPaneLayout

In the master-detail pattern, we are showing both the master and the detail fragment, side-by-side, on larger screens, while showing only one at a time on smaller screens. In the preceding examples, we had to manage all of that ourselves, in terms of deciding how many fragments to show and for switching between those fragments as needed.

`SlidingPaneLayout` encapsulates that logic.

`SlidingPaneLayout` will detect the screen size. If the screen size is big enough, `SlidingPaneLayout` will display its two children side-by-side. If the screen size is not big enough, `SlidingPaneLayout` will display one child at a time. However, by default, when the "master" child is visible, a thin strip on the right will allow the user to return to the "detail" child. Similarly, a swiping gesture can switch from the "detail" back to the "master" child. These are in addition to any changes in context you might introduce based on UI operations (e.g., tapping on an element in a master `ListView` automatically switching to the detail child).

Converting to SlidingPaneLayout

The `LargeScreen/EU4YouSlidingPane` sample project represents a rework of the EU4You core sample, this time using `SlidingPaneLayout` for handling the master-detail pattern.

Since `SlidingPaneLayout` encapsulates the master-detail logic, we can drop a lot of stuff that we used before but no longer need, including:

- `DetailsActivity` (as `SlidingPaneLayout` works akin to our single-activity implementation)
- the dedicated large-screen layout (as `SlidingPaneLayout` "bakes in" the logic for handling different screen sizes)
- dynamic fragments (as `SlidingPaneLayout` will work better with static fragments, anyway)
- `isPersistentSelection()` (as we will always want to use activated rows, on API Level 11+, as the user can more readily switch back and forth between

master and detail on smaller screens, and we want to indicate in the master what the context is that is displayed in the detail)

However, we did have to add a bit of pane management, plus move around some list-related behaviors in our `CountriesFragment`.

For starters, our `res/layout/main.xml` file now contains a `SlidingPaneLayout`, along with our two fragments, each set up as static `<fragment>` elements:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.SlidingPaneLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/panes"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <fragment
    android:id="@+id/countries"
    android:name="com.commonsware.android.eu4you4.CountriesFragment"
    android:layout_width="300sp"
    android:layout_height="match_parent"/>

  <fragment
    android:id="@+id/details"
    android:name="com.commonsware.android.eu4you4.DetailsFragment"
    android:layout_width="400dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

</android.support.v4.widget.SlidingPaneLayout>
```

(from LargeScreen/EU4YouSlidingPane/app/src/main/res/layout/main.xml)

By putting an `android:layout_weight` on our `details` fragment, we indicate that we want that one to take up all remaining room when the two fragments are shown side-by-side. You might think that we should then set the width of the `details` fragment to `0dp`; however, for some reason, this does not work.

The size of the `countries` (master) fragment will be honored on larger screens. On smaller screens, the size of the master fragment will be dictated by the width of the screen, minus a strip to allow the user to see a portion of the detail fragment and swipe that to display the detail fragment *in toto*.

Our `CountriesFragment` now always sets up the `ListView` to be single-choice mode, in `onActivityCreated()`. It also calls `onCountrySelected()` on our `CountriesFragment.Contract`, to ensure that the master is highlighting the last selection — this is needed to make sure that everything is displayed properly after a configuration change:

```
  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    setListAdapter(new CountryAdapter());
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);

    if (state != null) {
      int position=state.getInt(STATE_CHECKED, -1);

      if (position > -1) {
        getListView().setItemChecked(position, true);
        getContract().onCountrySelected(Country.EU.get(position));
      }
    }
  }
```

(from LargeScreen/EU4YouSlidingPane/app/src/main/java/com/commonsware/android/eu4you4/CountriesFragment.java)

`onListItemClick()` of `CountriesFragment` becomes a bit simpler:

```
  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    l.setItemChecked(position, true);
    getContract().onCountrySelected(Country.EU.get(position));
  }
```

(from LargeScreen/EU4YouSlidingPane/app/src/main/java/com/commonsware/android/eu4you4/CountriesFragment.java)

The `EU4You` activity overall becomes *substantially* simpler:

```
package com.commonsware.android.eu4you4;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.widget.SlidingPaneLayout;

public class EU4You extends Activity implements
    CountriesFragment.Contract {
  private DetailsFragment details=null;
  private SlidingPaneLayout panes=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    details=
```

```
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);
    panes=(SlidingPaneLayout)findViewById(R.id.panes);
    panes.openPane();
  }

  @Override
  public void onBackPressed() {
    if (panes.isOpen()) {
      super.onBackPressed();
    }
    else {
      panes.openPane();
    }
  }

  @Override
  public void onCountrySelected(Country c) {
    details.loadUrl(getString(c.url));
    panes.closePane();
  }
}
```

(from LargeScreen/EU4YouSlidingPane/app/src/main/java/com/commonsware/android/eu4you4/EU4You.java)

In `SlidingPaneLayout` terminology, the pane is "open" if the *master* is shown on smaller screens, and the pane is "closed" if the *detail* is shown on smaller screens. If this feels a bit counter-intuitive to you, you are not alone in that regard.

By default, the `SlidingPaneLayout` is closed. So, if we want to start (on smaller screens) with the master pane shown, we need to call `openPane()`, as we do in `onCreate()`. Similarly:

- If we want to show the details when the user clicks on a country in the `CountriesFragment`, we need to call `closePane()` in `onCountrySelected()`
- If we want to show the master pane if the user presses BACK while viewing the detail pane, we need to override `onBackPressed()` and consume that event (calling `openPane()`), instead of performing the normal superclass behavior

What SlidingPaneLayout Looks Like

On a larger screen, the `SlidingPaneLayout` edition of the `EU4You` activity looks the same as the prior examples.

However, on a smaller screen, things look slightly different. Specifically:

- Our master perspective has a thin strip on the right, showing a peek of the detail fragment

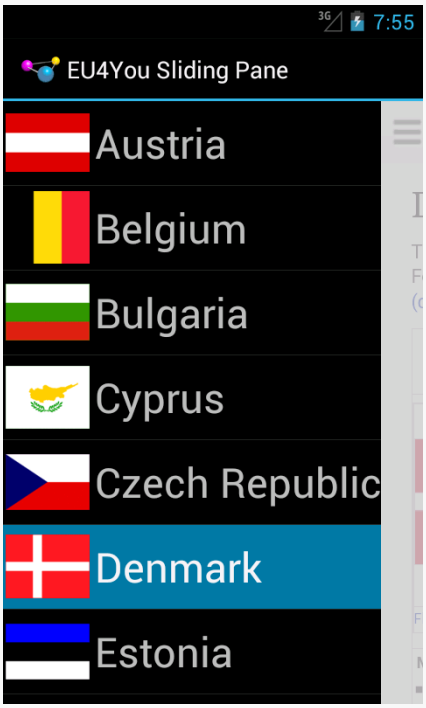


Figure 322: EU4YouSlidingPane, On a Phone Emulator, Showing Master

- The user can switch to the detail pane either by swiping open the detail pane or clicking on a country
- The user can switch back to the master pane either by swiping the detail pane back closed or by pressing the BACK button

Showing More Pages

`ViewPager` is a popular container in Android, as horizontal swiping is an increasingly popular navigational model, to move between peer pieces of content (e.g., swiping between contacts, swiping between book chapters). In some cases, when the `ViewPager` is on a larger screen, we simply want larger pages — a digital book reader, for example, would simply have a larger page in a bigger font for easier reading.

Sometimes, though, we might not be able to take advantage of the full space offered by the large screen, particularly when our `ViewPager` takes up the whole screen. In cases like this, it might be useful to allow `ViewPager`, in some cases, to show more than one page at a time. Each "page" is then designed to be roughly phone-sized, and we choose whether to show one, two, or perhaps more pages at a time based upon the available screen space.

Mechanically, allowing `ViewPager` to show more than one page is fairly easy, involving overriding one more method in our `PagerAdapter`: `getPageWidth()`. To see this in action, take a look at the `ViewPager/MultiView1` sample project.

Each page in this sample is simply a `TextView` widget, using the activity's style's "large appearance", centered inside a `LinearLayout`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center"
  android:orientation="vertical">

  <TextView
    android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

</LinearLayout>
```

(from ViewPager/MultiView1/app/src/main/res/layout/page.xml)

The activity, in `onCreate()`, gets our `ViewPager` from the `res/layout/activity_main.xml` resource, and sets its adapter to be a `SampleAdapter`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    pager=(ViewPager)findViewById(R.id.pager);
    pager.setAdapter(new SampleAdapter());
    pager.setOffscreenPageLimit(6);
  }
```

(from ViewPager/MultiView1/app/src/main/java/com/commonsware/android/mvp1/MainActivity.java)

In this case, `SampleAdapter` is not a `FragmentPagerAdapter`, nor a `FragmentStatePagerAdapter`. Instead, it is its own implementation of the `PagerAdapter` interface:

```
/*
 * Inspired by
 * https://gist.github.com/8cbe094bb7a783e37ad1
 */
private class SampleAdapter extends PagerAdapter {
  @Override
  public Object instantiateItem(ViewGroup container, int position) {
    View page=
        getLayoutInflater().inflate(R.layout.page, container, false);
    TextView tv=(TextView)page.findViewById(R.id.text);
    int blue=position * 25;

    final String msg=
        String.format(getString(R.string.item), position + 1);

    tv.setText(msg);
    tv.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
        Toast.makeText(MainActivity.this, msg, Toast.LENGTH_LONG)
             .show();
      }
    });

    page.setBackgroundColor(Color.argb(255, 0, 0, blue));
    container.addView(page);

    return(page);
  }

  @Override
  public void destroyItem(ViewGroup container, int position,
                          Object object) {
    container.removeView((View)object);
  }

  @Override
  public int getCount() {
    return(9);
  }

  @Override
  public float getPageWidth(int position) {
```

```
    return(0.5f);
  }

  @Override
  public boolean isViewFromObject(View view, Object object) {
    return(view == object);
  }
}
```

(from ViewPager/MultiView1/app/src/main/java/com/commonsware/android/mvp1/MainActivity.java)

To create your own `PagerAdapter`, the big methods that you need to implement are:

- `instantiateItem()`, where you create the page itself and add it to the supplied container. In this case, we inflate the page, set the text of the `TextView` based on the supplied position, set the background color of the page itself to be a different shade of blue based on the position, set up a click listener to show a `Toast` when the `TextView` is tapped, and use that for our page. We return some object that identifies this page; in this case, we return the inflated `View` itself. A fragment-based `PagerAdapter` would probably return the fragment.
- `destroyItem()`, where we need to clean up a page that is being removed from the pager, where the page is identified by the `Object` that we had previously returned from `instantiateItem()`. In our case, we just remove it from the supplied container.
- `isViewFromObject()`, where we confirm whether some specific page in the pager (represented by a `View`) is indeed tied to a specific `Object` returned from `instantiateItem()`. In our case, since we return the `View` from `instantiateItem()`, we merely need to confirm that the two objects are indeed one and the same.
- `getCount()`, as with the built-in `PagerAdapter` implementations, to return how many total pages there are.

In our case, we also override `getPageWidth()`. This indicates, for a given position, how much horizontal space in the `ViewPager` should be given to this particular page. In principle, each page could have its own unique size. The return value is a `float`, from `0.0f` to `1.0f`, indicating what fraction of the pager's width goes to this page. In our case, we return `0.5f`, to have each page take up half the pager.

The result is that we have two pages visible at a time:

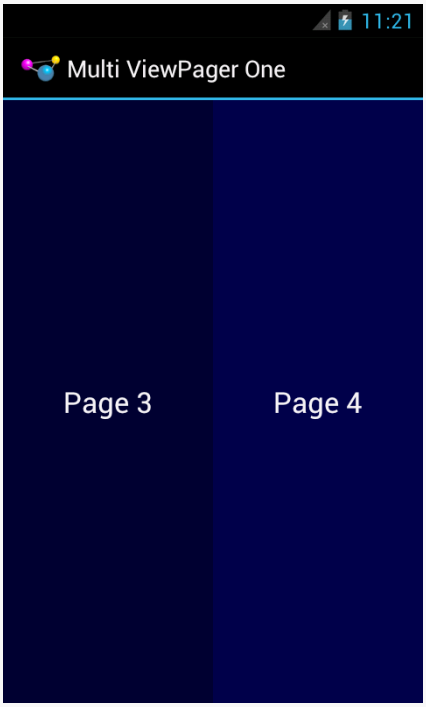


Figure 323: Two Pages in a ViewPager on Android 4.0.3

It is probably also a good idea to call `setOffscreenPageLimit()` on the `ViewPager`, as we did in `onCreate()`. By default (and at minimum), `ViewPager` will cache three pages: the one presently visible, and one on either side. However, if you are showing more than one at a time, you should bump the limit to be 3 times the number of simultaneous pages. For a page width of `0.5f` — meaning two pages at a time – you would want to call `setOffscreenPageLimit(6)`, to make sure that you had enough pages cached for both the current visible contents and one full swipe to either side.

`ViewPager` even handles "partial swipes" — a careful swipe can slide the right-hand page into the left-hand position and slide in a new right-hand page. And `ViewPager` stops when you run out of pages, so the last page will always be on the right, no matter how many pages at a time and how many total pages you happen to have.

The biggest downside to this approach is that it will not work well with the current crop of indicators. `PagerTitleStrip` and `PagerTabStrip` assume that there is a single selected page. While the indicator will adjust properly, the visual representation shows that the left-hand page is the one selected (e.g., the tab with the highlight), even though two or more pages are visible. You can probably overcome this with a custom indicator (e.g., highlight the selected tab and the one to its right).

Also note that this approach collides a bit with `setPageMargin()` on `ViewPager`. `setPageMargin()` indicates an amount of whitespace that should go in a gutter between pages. In principle, this would work great with showing multiple simultaneous pages in a `ViewPager`. However, `ViewPager` does not take the gutter into account when interpreting the `getPageWidth()` value. For example, suppose `getPageWidth()` returns `0.5f` and we `setPageMargin(20)`. On a 480-pixel-wide `ViewPager`, we will actually use 500 pixels: 240 for the left page, 240 for the right page, and 20 for the gutter. As a result, 20 pixels of our right-hand page are off the edge of the pager. Ideally, `ViewPager` would subtract out the page margin *before* applying the page width. One workaround is for you to derive the right `getPageWidth()` value based upon the `ViewPager` size and gutter yourself, rather than hard-coding a value. Or, build in your gutter into your page contents (e.g., using `android:layout_marginLeft` and `android:layout_marginRight`) and skip `setPageMargin()` entirely.

Columns or Pages

Another pattern — using pages for smaller screens and having the "pages" side-by-side in columns for larger screens — will be explored later in the book.

The Grid Pattern

Yet another approach for taking advantage of larger screen sizes is to always show a full-size master and a full-size detail — perhaps using different activities — but to use a grid rather than a list for the master. This works well when the data being shown in the grid can be represented as "cards", often dominated by some photo or other image.

The basic approach is to use fewer grid columns (e.g., 1 or 2) on smaller screen sizes and more grid columns (e.g., 3 or 4) on larger screen sizes. This way, the application flow is identical across screen sizes, yet the screen usage on larger screens is more effective. This is particularly true if you use on of the "staggered" grid widgets available from third parties, like Etsy's `AndroidStaggeredGrid` or Maurycy Wojtowicz's `StaggeredGridView`:

Figure 324: StaggeredGridView Demo (image courtesy of Maurycy Wojtowicz)

Fragment FAQs

Here are some other common questions about the use of fragments in support of large screen sizes:

Does *Everything* Have To Be In a Fragment?

In a word, no.

UI constructs that do not change based on screen size, configurations, and the like could simply be defined in the activity itself. For example, the activity can add items to the action bar that should be there regardless of what fragments are shown.

What If Fragments Are Not Right For Me?

While fragments are useful, they do not solve all problems. Few games will use fragments for the core of game play, for example. Applications with other forms of specialized user interfaces — painting apps, photo editors, etc. – may also be better served by avoiding fragments *for those specific activities* and doing something else.

That "something else" might start with custom layouts for the different sizes and orientations. At runtime, you can determine what you need either by inspecting what you got from the layout, or by using `Configuration` and `DisplayMetrics` objects to determine what the device capabilities are (e.g., screen size). The activity would then need to have its own code for handling whatever you want to do differently based on screen size (e.g., offering a larger painting canvas plus more on-screen tool palettes).

Do Fragments Work on TVs?

Much of the focus on "larger-screen devices" has been on tablets, because, as of the time of this writing, they are the most popular "larger-screen devices" in use. However, there are plenty of scenarios involving Android on TV to consider. A TV or other external display will present itself as a `-large` (720p) or `-xlarge` (1080p) screen. Fragments can certainly help with displaying a UI for a TV, but there are other design considerations to take into account, based upon the fact that the user sits much further from a TV than they do from a phone or tablet (so-called "10-foot user experience"). This is covered in greater detail later in the book.

Screen Size and Density Tactics

Even if we take the "tablet = several phones" design approach, the size of the "phone" will vary, depending on the size of the tablet. Plus, there are real actual *phones*, and those too vary in size. Hence, our fragments (or activities hosting their own UI directly) need to take into account micro fluctuations in size, as well as the macro ones.

Screen density is also something that affects us tactically. It is rare that an application will make wholesale UI changes based upon whether the screen is 160dpi or 240dpi or 320dpi or something else. However, changes in density can certainly impact the sizes of things, like images, that are intrinsically tied to pixel sizes. So, we need to take density into account as we are crafting our fragments to work well in a small range of sizes.

Dimensions and Units

As a unit of measure, the pixel (`px`) is a poor choice, because its size varies by density. Two phones might have very similar screen sizes but radically different densities. Anything specified in terms of pixels will be smaller on the higher-density device, and typically you would want them to be about the same size. For example, a

`Button` should not magically shrink for a ~4" phone just because the phone happens to have a much higher screen density than some other phone.

The best answer is to avoid specifying concrete sizes where possible. This is why you tend to see containers, and some widgets, use `match_parent` and `wrap_content` for their size — those automatically adjust based upon device characteristics.

Some places, though, you have to specify a more concrete size, such as with padding or margins. For these, you have two major groups of units of measure to work with:

- Those based upon pixels, but taking device characteristics into account. These include density-independent pixels (`dp` or `dip`), which try to size each `dp` to be about 1/160 of an inch. These also include scaled pixels (`sp`), which scales the size based upon the default font size on the device — `sp` is often used with `TextView` (and subclasses) for `android:textSize` attributes.
- Those based purely on physical units of measure: `mm` (millimeters), `in` (inches), and `pt` (points = 1/72 of an inch).

Any of those tends to be better than `px`. Which you choose will depend on which you and your graphics designer are more comfortable with.

If you find that there are cases where the dimensions you want to use vary more widely than the automatic calculations from these density-aware units of measure, you can use dimension resources. Create a `dimens.xml` file in `res/values/` and related resource sets, and put in there `<dimen>` elements that give a dimension a name and a size. In addition to perhaps making things a bit more DRY ("don't repeat yourself"), you can perhaps create different values of those dimensions for different screen sizes, densities, or other cases as needed.

Layouts and Stretching

Web designers need to deal with the fact that the user might resize their browser window. The approaches to deal with this are called "fluid" designs.

Similarly, Android developers need to create "fluid" layouts for fragments, rows in a `ListView`, and so on, to deal with similar minor fluctuations in size.

Each of "The Big Three" container classes has its approach for dealing with this:

- Use `android:layout_weight` with `LinearLayout` to allocate extra space

- Use `android:stretchColumns` and `android:shrinkColumns` with `TableLayout` to determine which columns should absorb extra space and which columns should be forcibly "shrunk" to yield space for other columns if we lack sufficient horizontal room
- Use appropriate rules on `RelativeLayout` to anchor widgets as needed to other widgets or the boundaries of the container, such that extra room flows naturally wherever the rules call for

Drawables That Resize

Images, particularly those used as backgrounds, will need to be resized to take everything into account:

- screen size and density
- size of the widget, and its contents, for which it serves as the background (e.g., amount of prose in a `TextView`)

Android supports what is known as the "nine-patch" PNG format, where resizing information is held in the PNG itself. This is typically used for things like rounded rectangles, to tell Android to stretch the straight portions of the rectangle but to not stretch the corners. Nine-patch PNG files will be examined in greater detail in a later chapter of this book.

The `ShapeDrawable` XML drawable resource uses an ever-so-tiny subset of SVG (Scalable Vector Graphics) to create a vector art definition of an image. Once again, this tends to be used for rectangles and rounded rectangles, particularly those with a gradient fill. Since Android interprets the vector art definition at runtime, it can create a smooth gradient, interpolating all intervening colors from start to finish. Stretching a PNG file — even a nine-patch PNG file — tends to result in "banding effects" on the gradients. `ShapeDrawable` is also covered later in this book.

Third-party libraries can also help. The `svg-android` project supplies a JAR that handles more SVG capabilities than does `ShapeDrawable`, though it too does not cover the entire SVG specification. Also, `WebView` has some ability to view SVG files on Android 3.0+.

Drawables By Density

Sometimes, though, there is no substitute for your traditional bitmap image. Icons and related artwork are not necessarily going to be stretched at runtime, but they are still dependent upon screen density. A 80x80 pixel image may look great on a

Samsung Galaxy Nexus or other `-xhdpi` device, coming in at around a quarter-inch on a side. However, when viewed on a `-mdpi` device, that same icon will be a half-inch on a side, which may be entirely too large.

The best answer is to create multiple renditions of the icon at different densities, putting each icon in the appropriate drawable resource directory (e.g., `res/drawable-mdpi`, `res/drawable-hdpi`). This is what Android Asset Studio did for us in the tutorials, creating launcher icons from some supplied artwork for all four densities. Even better is to create icons tailored for each density — rather than just reducing the pixel count, take steps to draw an icon that will still make sense to the user at the lower pixel count, exaggerating key design features and dropping other stuff off. Google's Kirill Grouchnikov has an excellent blog post on this aspect

However, Android will let you cheat.

If you supply only some densities, but your app runs on a device with a different density, Android will automatically resample your icons to try to generate one with the right density, to keep things the same size. On the plus side, this saves you work — perhaps you only ship an `-xhdpi` icon and let Android do the rest. And it can reduce your APK size by a bit. However, there are costs:

- This is a bit slower at runtime and consumes a bit more battery
- Android's resampling algorithm may not be as sophisticated as that of your preferred image editor (e.g., Photoshop)
- You cannot finesse the icon to look better than a simple resampling (e.g., drop off design elements that become unidentifiable)

Other Considerations

There are other things you should consider when designing your app to work on multiple screen sizes, beyond what is covered above.

Small-Screen Devices

It is easy to think of screen size issues as being "phones versus tablets". However, not only do tablets come in varying sizes (5" Samsung Galaxy Note to a bunch of 10.1" tablets), but phones come in varying sizes. Those that have less than a 3" diagonal screen size will be categorized as `-small` screen devices, and you can have different layouts for those.

Getting things to work on small screens is sometimes more difficult than moving from normal to larger screens, simply because you lack sufficient room. You can only shrink widgets so far before they become unreadable or "untappable". You may need to more aggressively use `ScrollView` to allow your widgets to have more room, but requiring the user to pan through your whole fragment's worth of UI. Or, you may need to divide your app into more fragments than you originally anticipated, and use more activities or other tricks to allow the user to navigate the fragments individually on small-screen devices, while stitching them together into larger blocks for larger phones.

Avoid Full-Screen Backgrounds

Android runs in lots of different resolutions.

Lots and lots of different resolutions.

Trying to create artwork for each and every resolution in use today will be tedious and fragile, the latter because new resolutions pop up every so often, ones you may not be aware of.

Hence, try to design your app to avoid some sort of full-screen background, where you are expecting the artwork to perfectly fit the screen. Either:

- Do not use a background, or
- Use a background, but one that is designed to be cropped to fit and will look good in its cropped state, or
- Use a background, but one that can naturally bleed into some solid fill to the edges (e.g., a starfield that simply lacks stars towards the edges), so you can "fill in" space around your background with that solid color to fill the screen, or
- Dynamically draw the background (e.g., a starfield where you place the stars yourself at runtime using 2D graphics APIs)

For most conventional apps, just using the background from your stock theme will typically suffice. This problem is much bigger for 2D games, which tend to rely upon backgrounds as a game surface.

Manifest Elements for Screen Sizes

There are two elements you can add to your manifest that impact how your application will behave with respect to screen sizes.

<compatible-screens> serves as an advertisement of your capabilities, to the Google Play Store and similar "markets". You can have a <compatible-screens> element with one or more child <screen> elements — each <screen> enumerates a combination of screen size and screen density that you support:

```
<compatible-screens>
    <!-- all possible normal size screens -->
    <screen android:screenSize="normal" android:screenDensity="ldpi" />
    <screen android:screenSize="normal" android:screenDensity="mdpi" />
    <screen android:screenSize="normal" android:screenDensity="hdpi" />
    <screen android:screenSize="normal" android:screenDensity="xhdpi" />
    <!-- all possible large size screens -->
    <screen android:screenSize="large" android:screenDensity="ldpi" />
    <screen android:screenSize="large" android:screenDensity="mdpi" />
    <screen android:screenSize="large" android:screenDensity="hdpi" />
    <screen android:screenSize="large" android:screenDensity="xhdpi" />
</compatible-screens>
```

The Google Play Store will filter your app, so it will not show up on devices that have screens that do not meet one of your <screen> elements. However, new densities show up every year or so, and devices running those densities will *not* be supported by your <compatible-screens> element unless you add the appropriate <screen> element for that density. For example, the above <compatible-screens> element does not cover tvdpi or xxhdpi devices. As a result, Google discourages the use of <compatible-screens>.

There is also a <supports-screens> element, as we saw when we set up our initial project in the tutorials. Here, you indicate what screen sizes you support, akin to <compatible-screens> (minus any density declarations). And, the Google Play Store will filter your app, so it will not show up on devices that have screens *smaller than what you support*.

So, for example, suppose that you have a <supports-screens> element like this:

```
<supports-screens android:smallScreens="false"
                  android:normalScreens="true"
                  android:largeScreens="true"
                  android:xlargeScreens="false"
/>
```

You will not show up in the Google Play Store for any -small screen devices. However, you *will* show up in the Google Play Store for any -xlarge screen devices — Android will merely apply some runtime logic to try to help your app run well on

such screens. So, while <compatible-screens> is purely a filter, <supports-screens> is a filter for smaller-than-supported screens, and a runtime "give me a hand!" flag for larger-than-supported screens.

Considering Newer Densities

-tvdpi — around 213dpi — was added for Android TV, and is the density used for 720p Android TV devices. However, Google *also* elected to use -tvdpi for the Nexus 7 tablet. However, not even Google bothered to create many -tvdpi-specific resources, allowing the OS to downsample from the -hdpi edition.

-xxhdpi was added in late 2012 and is for devices with a screen density around 480dpi. While Android can up-sample an -xhdpi image for -xxhdpi, the results may not be as crisp as you would like. Hence, you may wish to consider creating -xxhdpi as your "top tier" density, so other devices can downsample if needed. At the time of this writing, about 15% of Play Store-equipped Android devices are -xxhdpi.

-xxxhdpi is for devices with screens around 640dpi. At the time of this writing -xxxhdpi is not in significant use.

Tutorial #17 - Supporting Large Screens

So far, we have created a variety of fragments that are being used one at a time in a hosting activity: notes, help, and about. And, on smaller-screen devices, like phones, that is probably the best solution. But on devices like 10" tablets, it might be nice to be able to have some of those fragments take over a part of the main activity's space. For example, the user could be reading the chapter and reading the online help.

Hence, in this tutorial, we will arrange for the help and about fragments to be loaded into `EmPubLiteActivity` directly on tablets, while retaining our existing functionality for other devices.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository.

Step #1: Creating Our Layouts

The simplest way to both add a place for these other fragments and to determine when we should be using these other fragments in the main activity is to create new `layout` resource sets for larger-screen devices, with customized versions of `main.xml` to be used by `EmPubLiteActivity`.

Right-click over the `res/` directory in your app, then choose New > "Android Resource Directory" from the context menu. As before, this brings up the new resource directory dialog:

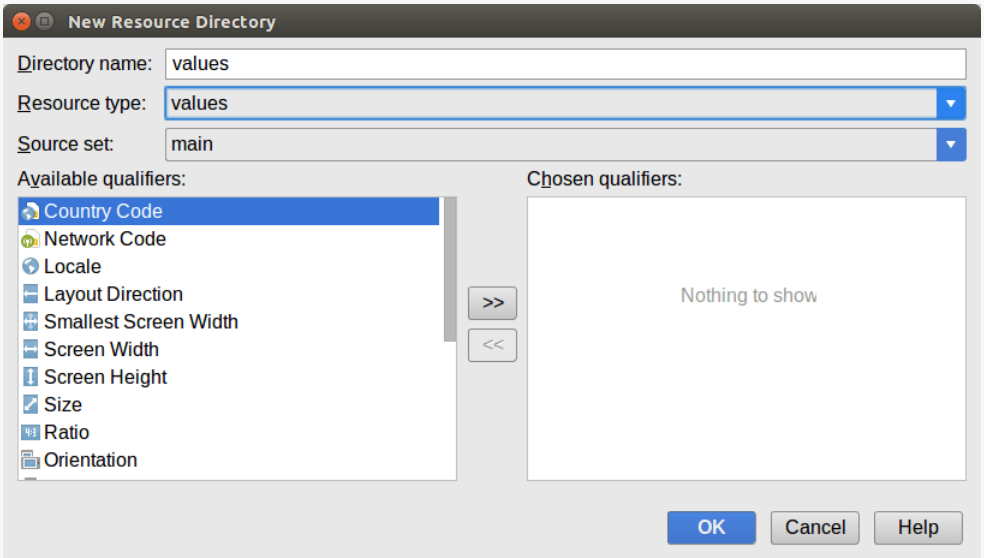


Figure 325: Android Studio New Resource Dialog, As Initially Opened

Choose "layout" from the "Resource type" drop-down. Then, click on "Screen Width" in the list of qualifiers on the left, and click the ">>" button to add that to the list on the right:

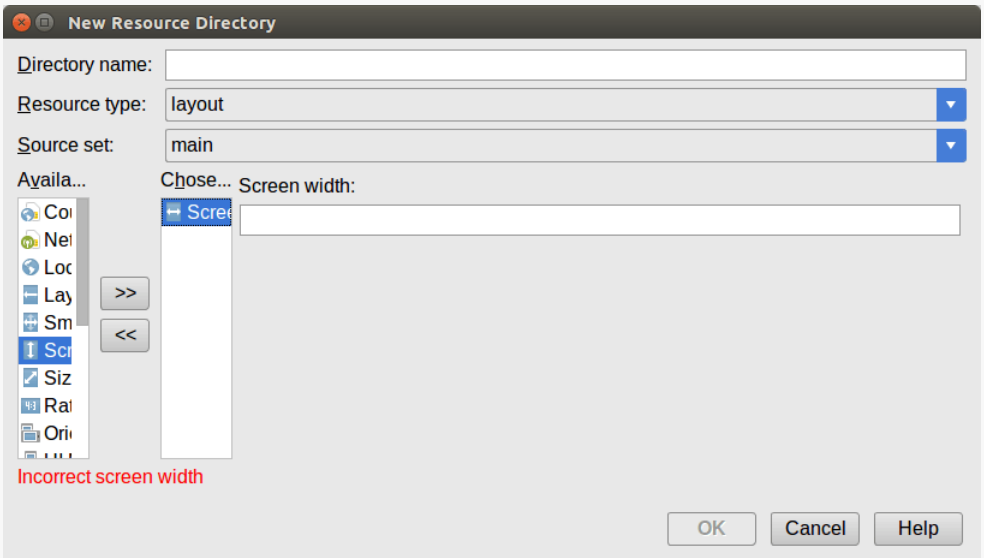


Figure 326: Android Studio New Resource Dialog, After Selecting "Screen Width"

In the "Screen width" field, fill in `880`:

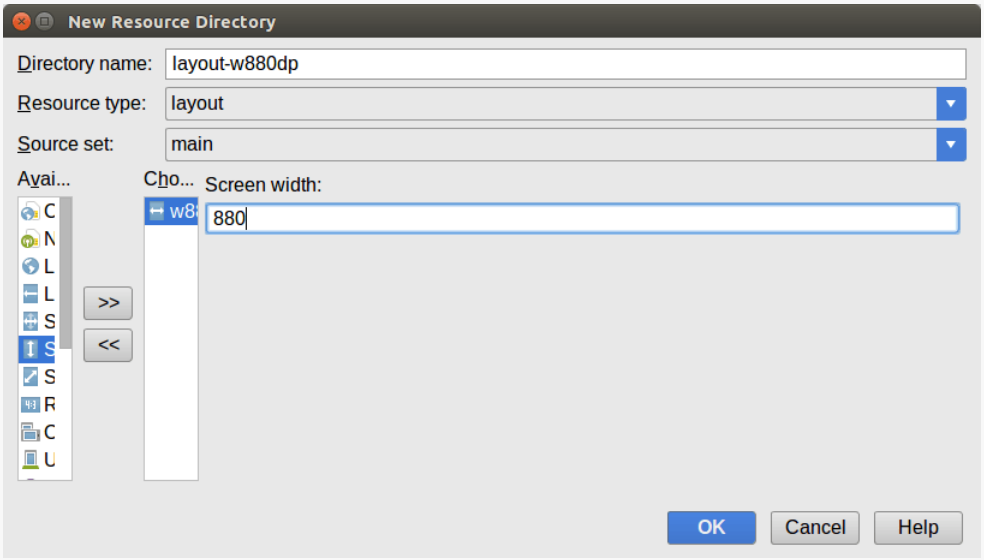


Figure 327: Android Studio New Resource Dialog, After Setting Screen Width

Click OK to create the directory. Repeat that process to create a `res/layout-h880dp/` directory, this time choosing "Screen Height" rather than "Screen Width".

Then, right-click over the `res/layout/main.xml` file and choose "Copy" from the context menu. After that, right-click over the new `res/layout-w880dp/` directory and choose "Paste" from the context menu. This brings up the copy dialog:

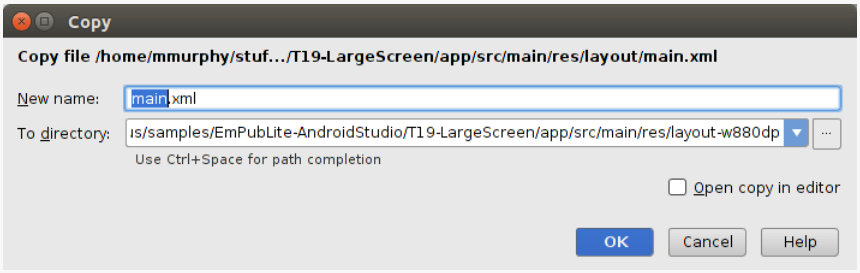


Figure 328: Android Studio Copy Dialog

Check the "Open copy in editor" checkbox and click OK. This will bring up the graphical layout editor on this copy of the `main` layout.

Unfortunately, what we want to do is not readily supported by Android Studio's edition of the drag-and-drop GUI builder. So, switch over to the XML for this layout, and replace it with:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">

  <LinearLayout
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="7"
    android:orientation="vertical">

    <io.karim.MaterialTabs
      android:id="@+id/tabs"
      android:layout_width="match_parent"
      android:layout_height="48dp"
      app:mtIndicatorColor="@color/colorAccent"
      app:mtSameWeightTabs="true" />

    <android.support.v4.view.ViewPager
      android:id="@+id/pager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"></android.support.v4.view.ViewPager>
  </LinearLayout>

  <View
    android:id="@+id/divider"
    android:layout_width="2dp"
    android:layout_height="match_parent"
    android:background="#AA000000"
    android:visibility="gone" />

  <FrameLayout
    android:id="@+id/sidebar"
    android:layout_width="0dp"
    android:layout_height="match_parent" />
</LinearLayout>
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/res/layout-w880dp/main.xml)

Repeat the same process, copying `res/layout/main.xml` into the `res/layout-h880dp/` directory, and replacing the copy's contents with:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
```

```
  android:layout_height="match_parent"
  android:orientation="vertical">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="7"
    android:orientation="vertical">

    <io.karim.MaterialTabs
      android:id="@+id/tabs"
      android:layout_width="match_parent"
      android:layout_height="48dp"
      app:mtIndicatorColor="@color/colorAccent"
      app:mtSameWeightTabs="true" />

    <android.support.v4.view.ViewPager
      android:id="@+id/pager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"></android.support.v4.view.ViewPager>
  </LinearLayout>

  <View
    android:id="@+id/divider"
    android:layout_width="match_parent"
    android:layout_height="2dp"
    android:background="#AA000000"
    android:visibility="gone" />

  <FrameLayout
    android:id="@+id/sidebar"
    android:layout_width="match_parent"
    android:layout_height="0dp"></FrameLayout>
</LinearLayout>
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/res/layout-h880dp/main.xml)

Step #2: Loading Our Sidebar Widgets

Now that we added the divider widget and sidebar container to (some of) our layouts, we need to access those widgets at runtime.

So, in `EmPubLiteActivity`, add data members for them:

```
  private View sidebar=null;
  private View divider=null;
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Then, in `onCreate()` of `EmPubLiteActivity`, initialize those data members, sometime after the call to `setContentView()`:

```
    sidebar=findViewById(R.id.sidebar);
    divider=findViewById(R.id.divider);
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Step #3: Opening the Sidebar

A real production-grade app would use animated effects to hide and show our sidebar. However, we have not yet covered animations in this book, so we will simply:

- Cause the divider to become visible
- Adjust the `android:layout_weight` of our sidebar to be `3` instead of `0`, giving it ~30% of the screen (with the original `RelativeLayout` getting 70%, courtesy of its `android:layout_weight="7"`)

With that in mind, add the following implementation of an `openSidebar()` method to `EmPubLiteActivity`:

```
  private void openSidebar() {
    LinearLayout.LayoutParams p=
      (LinearLayout.LayoutParams)sidebar.getLayoutParams();
    if (p.weight == 0) {
      p.weight=3;
      sidebar.setLayoutParams(p);
    }

    divider.setVisibility(View.VISIBLE);
  }
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Here, we:

- Get the existing `LinearLayout.LayoutParams` from the sidebar

- If it is still `0` (meaning the sidebar has not been opened), assign it a weight of `3`, update the layout via `setLayoutParams()`, and toggle the visibility of the divider

Step #4: Loading Content Into the Sidebar

Now that we can get our sidebar to appear, we need to load content into it... but only if we have the sidebar. If `EmPubLiteActivity` loads a layout that does not have the sidebar, we need to stick with our existing logic that starts up an activity to display the content.

With that in mind, add data members to `EmPubLiteActivity` to hold onto our help and about fragments:

```
private SimpleContentFragment help=null;
private SimpleContentFragment about=null;
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Also add a pair of static data members that will be used as tags for identifying these fragments in our `FragmentManager`:

```
private static final String HELP="help";
private static final String ABOUT="about";
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Also add a pair of static data members that will hold the paths to our help and about assets, since we will be referring to them from more than one place when we are done:

```
private static final String FILE_HELP=
  "file:///android_asset/misc/help.html";
private static final String FILE_ABOUT=
  "file:///android_asset/misc/about.html";
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

In `onCreate()` of `EmPubLiteActivity`, initialize the fragments from the `FragmentManager`:

```
    help=(SimpleContentFragment)getFragmentManager().findFragmentByTag(HELP);
    about=
      (SimpleContentFragment)getFragmentManager().findFragmentByTag(ABOUT);
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

The net result is that *if* we are returning from a configuration change, we will have our fragments, otherwise we will not at this point.

Next, add the following methods to `EmPubLiteActivity`:

```
private void showAbout() {
  if (sidebar!=null) {
    openSidebar();

    if (about==null) {
      about=SimpleContentFragment.newInstance(FILE_ABOUT);
    }

    getFragmentManager().beginTransaction().addToBackStack(null)
      .replace(R.id.sidebar, about, ABOUT).commit();
  }
  else {
    Intent i=new Intent(this, SimpleContentActivity.class);

    i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_ABOUT);
    startActivity(i);
  }
}

private void showHelp() {
  if (sidebar!=null) {
    openSidebar();

    if (help==null) {
      help=SimpleContentFragment.newInstance(FILE_HELP);
    }

    getFragmentManager().beginTransaction().addToBackStack(null)
      .replace(R.id.sidebar, help, HELP).commit();
  }
  else {
    Intent i=new Intent(this, SimpleContentActivity.class);

    i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_HELP);
    startActivity(i);
```

```
    }
  }
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Both of these methods follow the same basic recipe:

- Check to see if `sidebar` is `null`, to see if we have a sidebar or not
- If we have a sidebar, call `openSidebar()` to ensure the user can see the sidebar, create our `Fragment` if we do not already have it, and use a `FragmentTransaction` to replace whatever was in the sidebar with the new `Fragment`
- If we do not have the sidebar, launch an activity with an appropriately-configured `Intent`

Note a couple of things with our `FragmentTransaction` objects:

- We use `addToBackStack(null)`, so if the user presses BACK, Android will reverse this transaction
- We use `replace()` instead of `add()`, as there may already be a fragment in the sidebar (`replace()` will behave the same as `add()` for an empty sidebar)

Then, in the `onOptionsItemSelected()` of `EmPubLiteActivity`, replace the `about`, and `help` `case` blocks to use the newly-added methods, replacing their existing implementations:

```
      case R.id.about:
        showAbout();

        return(true);

      case R.id.help:
        showHelp();

        return(true);
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Step #5: Removing Content From the Sidebar

While `addToBackStack(null)` will allow Android to automatically remove fragments as the user presses BACK, that will not cause our sidebar to magically close. Rather, we need to do that ourselves.

The easiest way to track this is to track the state of the "back stack". So, add `implements FragmentManager.OnBackStackChangedListener` to the declaration of `EmPubLiteActivity`, and in `onCreate()` of `EmPubLiteActivity`, add the following line, sometime after you initialized the `sidebar` and `divider` data members:

```
    getFragmentManager().addOnBackStackChangedListener(this);
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

This statement registers our activity as receiving events related to changes in the state of the back stack.

To make this compile, we need to implement `onBackStackChanged()` in `EmPubLiteActivity`:

```
  @Override
  public void onBackStackChanged() {
    if (getFragmentManager().getBackStackEntryCount() == 0) {
      LinearLayout.LayoutParams p=
        (LinearLayout.LayoutParams)sidebar.getLayoutParams();
      if (p.weight > 0) {
        p.weight=0;
        sidebar.setLayoutParams(p);
        divider.setVisibility(View.GONE);
      }
    }
  }
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

Here, if our back stack is empty, we reverse the steps from `openSidebar()` and close it back up again, hiding the divider and setting the sidebar's weight to `0`.

The complete revised `EmPubLiteActivity` should now look something like:

```
package com.commonsware.empublite;
```

```
import android.app.Activity;
import android.app.FragmentManager;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.os.StrictMode;
import android.support.v4.view.ViewPager;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.LinearLayout;
import org.greenrobot.eventbus.EventBus;
import org.greenrobot.eventbus.Subscribe;
import org.greenrobot.eventbus.ThreadMode;
import io.karim.MaterialTabs;

public class EmPubLiteActivity extends Activity
  implements FragmentManager.OnBackStackChangedListener {
  private static final String MODEL="model";
  private static final String PREF_LAST_POSITION="lastPosition";
  private static final String PREF_SAVE_LAST_POSITION="saveLastPosition";
  private static final String PREF_KEEP_SCREEN_ON="keepScreenOn";
  private static final String HELP="help";
  private static final String ABOUT="about";
  private static final String FILE_HELP=
    "file:///android_asset/misc/help.html";
  private static final String FILE_ABOUT=
    "file:///android_asset/misc/about.html";
  private ViewPager pager;
  private ContentsAdapter adapter;
  private ModelFragment mfrag=null;
  private View sidebar=null;
  private View divider=null;
  private SimpleContentFragment help=null;
  private SimpleContentFragment about=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    setupStrictMode();
    pager=(ViewPager)findViewById(R.id.pager);
    sidebar=findViewById(R.id.sidebar);
    divider=findViewById(R.id.divider);
    help=(SimpleContentFragment)getFragmentManager().findFragmentByTag(HELP);
    about=
      (SimpleContentFragment)getFragmentManager().findFragmentByTag(ABOUT);
```

```
    getFragmentManager().addOnBackStackChangedListener(this);
  }

  @Override
  public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);

    if (adapter==null) {
      mfrag=(ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

      if (mfrag==null) {
        mfrag=new ModelFragment();

        getFragmentManager().beginTransaction()
          .add(mfrag, MODEL).commit();
      }
      else if (mfrag.getBook()!=null) {
        setupPager(mfrag.getBook());
      }
    }

    if (mfrag.getPrefs()!=null) {
      pager.setKeepScreenOn(mfrag.getPrefs()
        .getBoolean(PREF_KEEP_SCREEN_ON, false));
    }
  }

  @Override
  public void onStop() {
    EventBus.getDefault().unregister(this);

    if (mfrag.getPrefs()!=null) {
      int position=pager.getCurrentItem();

      mfrag.getPrefs().edit().putInt(PREF_LAST_POSITION, position)
        .apply();
    }

    super.onStop();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);

    return(super.onCreateOptionsMenu(menu));
  }
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.about:
      showAbout();

      return(true);

    case R.id.help:
      showHelp();

      return(true);

    case R.id.settings:
      startActivity(new Intent(this, Preferences.class));

      return(true);

    case R.id.notes:
      startActivity(new Intent(this, NoteActivity.class)
        .putExtra(NoteActivity.EXTRA_POSITION,
          pager.getCurrentItem()));

      return(true);

    case R.id.update:
      startService(new Intent(this, DownloadCheckService.class));

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}

@Override
public void onBackStackChanged() {
  if (getFragmentManager().getBackStackEntryCount() == 0) {
    LinearLayout.LayoutParams p=
      (LinearLayout.LayoutParams)sidebar.getLayoutParams();
    if (p.weight > 0) {
      p.weight=0;
      sidebar.setLayoutParams(p);
      divider.setVisibility(View.GONE);
    }
  }
}
```

```
@SuppressWarnings("unused")
@Subscribe(threadMode =ThreadMode.MAIN)
public void onBookLoaded(BookLoadedEvent event) {
  setupPager(event.getBook());
}

private void setupPager(BookContents contents) {
  adapter=new ContentsAdapter(this, contents);
  pager.setAdapter(adapter);

  MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
  tabs.setViewPager(pager);

  SharedPreferences prefs=mfrag.getPrefs();

  if (prefs!=null) {
    if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
      pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
    }

    pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
  }
}

private void setupStrictMode() {
  StrictMode.ThreadPolicy.Builder builder=
    new StrictMode.ThreadPolicy.Builder()
      .detectAll()
      .penaltyLog();

  if (BuildConfig.DEBUG) {
    builder.penaltyFlashScreen();
  }

  StrictMode.setThreadPolicy(builder.build());
}

private void openSidebar() {
  LinearLayout.LayoutParams p=
    (LinearLayout.LayoutParams)sidebar.getLayoutParams();
  if (p.weight == 0) {
    p.weight=3;
    sidebar.setLayoutParams(p);
  }

  divider.setVisibility(View.VISIBLE);
}
```

```
  private void showAbout() {
    if (sidebar!=null) {
      openSidebar();

      if (about==null) {
        about=SimpleContentFragment.newInstance(FILE_ABOUT);
      }

      getFragmentManager().beginTransaction().addToBackStack(null)
        .replace(R.id.sidebar, about, ABOUT).commit();
    }
    else {
      Intent i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_ABOUT);
      startActivity(i);
    }
  }

  private void showHelp() {
    if (sidebar!=null) {
      openSidebar();

      if (help==null) {
        help=SimpleContentFragment.newInstance(FILE_HELP);
      }

      getFragmentManager().beginTransaction().addToBackStack(null)
        .replace(R.id.sidebar, help, HELP).commit();
    }
    else {
      Intent i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_HELP);
      startActivity(i);
    }
  }
}
```

(from EmPubLite-AndroidStudio/T17-LargeScreen/EmPubLite/app/src/main/java/com/commonsware/empublite/EmPubLiteActivity.java)

At this point, if you build the project and run it on a sufficiently-large device or emulator, and you choose to view the help or about pages, you will see the sidebar appear, whether in portrait or landscape.

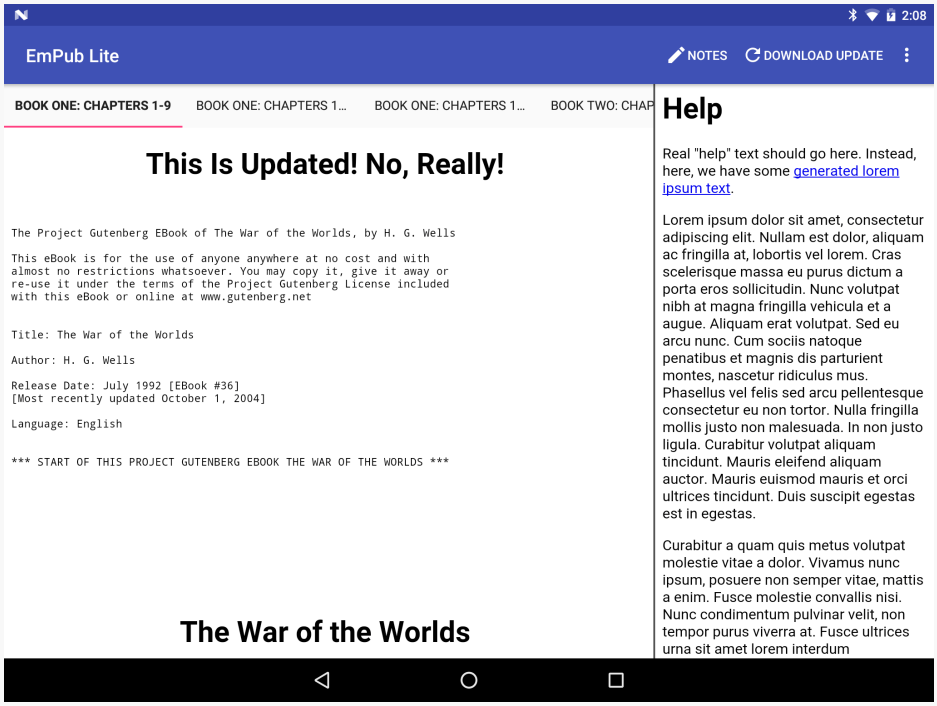


Figure 329: EmPubLite, on a Tablet-Sized Emulator, With Help

Note that a tablet emulator usually will only run acceptably fast if you are using the x86 emulator images.

Backwards Compatibility Strategies and Tactics

Android is an ever-moving target, averaging about 2.5 API level increments per year. The Android Developer site maintains a chart and table showing the most recent breakdown of OS versions making requests of the Play Store.

Most devices tend to be clustered around 1-3 minor releases. However, these are never the most recent release, which takes time to percolate through the device manufacturers and carriers and onto devices, whether those are new sales or upgrades to existing devices.

Some developers panic when they realize this.

Panic is understandable, if not necessary. This is a well-understood problem, that occurs frequently within software development — ask any Windows developer who had to simultaneously support everything from Windows 98 to Windows XP, or Windows XP through Windows 8.1. Moreover, there are many things in Android designed to make this problem as small as possible. What you need are the strategies and tactics to make it all work out.

Think Forwards, Not Backwards

Android itself tries very hard to maintain backwards compatibility. While each new Android release adds many classes and methods, relatively few are marked as deprecated, and almost none are outright eliminated. And, in Android, "deprecated" means "there's probably a better solution for what you are trying to accomplish, though we will maintain this option for you as long as we can".

Despite this, many developers aim purely for the lowest common denominator. Aiming to support older releases is noble. Ignoring what has happened since those releases is stupid, if you are trying to distribute your app to the public via the Play Store or similar mass-distribution means.

Why? You want your app to be distinctive, not decomposing.

For example, as we saw in the chapter on the action bar, adding one line to the manifest (`android:targetSdkVersion="11"`) gives you the action bar, the holographic widget set (e.g., `Theme.Holo`), the new style of options menu, and so on. Those dead-set on avoiding things newer than Android 2.1 would not use this attribute. As a result, on Android 3.0+ devices, their apps will tend to look old. Some will not, due to other techniques they are employing (e.g., running games in a full-screen mode), but many will.

You might think that this would not matter. After all, how many people in 2011 were even using Android 3.x? 5%?

However, those in position to trumpet your application — Android enthusiast bloggers chief among them — will tend to run newer equipment. Their opinion matters, if you are trying to have their opinion sway others relative to your app. Hence, if you look out-of-touch to them, they may be less inclined to provide glowing recommendations of your app to their readers.

Besides, not everything added to newer versions of Android is pure "eye candy". It is entirely possible that features in the newer Android releases might help make your app stand out from the competition, whether it is making greater use of NFC or offering tighter integration to the stock Calendar application or whatever. By taking an "old features only" approach, you leave off these areas for improvement.

And, to top it off, the world moves faster than you think. It takes about a year for a release to go from release to majority status (or be already on the downslope towards oblivion, passed over by something newer still). You need to be careful that the decisions you make today do not doom you tomorrow. If you focus on "old features only", how much rework will it take you to catch up in six months, or a year?

Hence, this book advocates an approach that differs from that taken by many: aim high. Decide what features you want to use, whether those features are from older releases or the latest-and-greatest release. Then, write your app using those features, and take steps to ensure that everything still works reasonably well (if not as full-featured) on older devices. This too is a well-trodden path, used by Web developers

for ages (e.g., support sexy stuff in Firefox and Safari, while still gracefully degrading for IE6). And the techniques that those Web developers use have their analogous techniques within the Android world.

Aim Where You Are Going

One thing to bear in mind is that the OS distribution chart and table shown above is based on devices contacting the Play Store. Hence, this is only directly relevant if you are actually distributing through the Play Store.

If you are distributing through the Amazon AppStore, or to device-specific outlets (e.g., BlackBerry World), you will need to take into account what sorts of devices are using those means of distribution.

If you are specifically targeting certain non-Play Store devices, like the Kindle Fire, you will need to take into account what versions of Android they run.

If you are building an app to be distributed by a device manufacturer on a specific device, you need to know what Android version will (initially) be on that device and focus on it.

If you are distributing your app to employees of a firm, members of an organization, or the like, you need to determine if there is some specific subset of devices that they use, and aim accordingly. For example, some enterprises might distribute Android devices to their employees, in which case apps for that enterprise should run on those devices, not necessarily others.

A Target-Rich Environment

There are a few places in your application where you will need to specify Android API levels of relevance to your code.

The most important one is the `android:minSdkVersion` attribute, as discussed early in this book. You need to set this to the oldest version of Android you are willing to support, so you will not be installed on devices older than that.

There is also `android:targetSdkVersion`, mentioned in passing earlier in this chapter. In the abstract, this attribute tells Android "this is the version of Android I was thinking of when I wrote the code". Android can use this information to help both backwards and forwards compatibility. Historically, this was under-utilized.

However, with API Level 11 and API Level 14, `android:targetSdkVersion` took on greater importance. Specifying 11 or higher gives you the action bar and all the rest of the look-and-feel introduced in the Honeycomb release. Specifying 14 or higher will give you some new features added in Ice Cream Sandwich, such as automatic whitespace between your app widgets and other things on the user's home screen. In general, use a particular `android:targetSdkVersion` when instructions tell you to.

The third place — and perhaps the one that confuses developers the most – is the build target. This shows up as `compileSdkVersion` in `build.gradle` for Android Studio and Gradle users.

Part of the confusion is the multiple uses of the term "target". The build target has nothing to do with `android:targetSdkVersion`. Nor is it strictly tied to what devices you are targeting.

Rather, it is a very literal term: it is the target **of the build**. It indicates:

- What version of the Android class library you wish to compile against, dictating what classes and methods you will be able to refer to directly
- What rules to apply when interpreting resources and the manifest, to complain about things that are not recognized

The net is that you set your build target to be the lowest API level that has everything you are using directly.

Lint: It's Not Just For Belly Buttons

In the old days, the only way to find out that you were using a newer class or method than what was in your `minSdkVersion` would be to set your build target to be the same as your `minSdkVersion`. That way, any attempt to use something newer than your minimum would be greeted with compile errors. This works, but at a high cost: it makes *intentionally* using newer capabilities very painful, forcing you to use reflection to access them.

Nowadays, this is no longer needed, thanks to Lint.

Lint is part of the standard build process, adding new errors and warnings for things that are syntactically valid but probably not the right answer. In particular, Lint will tell you if you are using classes or methods that are newer than your `minSdkVersion`, *even if they are valid for your build target*.

Hence, the targeting strategy nowadays is:

- Set your `minSdkVersion` to be the oldest version that you are willing to support
- Set your build target to be the version of Android that has all of the classes and methods you intend to use, allowing Lint to point out places where you need to pay attention to what sort of device you are running on (more on this later)
- Set your `targetSdkVersion` to be something relatively recent, unless you have specific reasons to use some specific version

A Little Help From Your Friends

The simplest way to use a feature yet support devices that lack the feature is to use a compatibility library that enables the feature for more devices. The Android Support package is one such compatibility library, though it also offers other classes as well.

With a compatibility library, the API for using the library is nearly identical to using the native Android capability, mostly involving slightly different package names (e.g., `android.support.v4.app.Fragment` instead of `android.app.Fragment`).

So, if there is something new that you want to use on older devices, and the new feature is not obviously tied to hardware, see if there is a "backport" of the feature available to you. Examples include backports of:

- `CalendarView` (https://github.com/SimonVT/android-calendarview)
- `Switch` (https://github.com/BoD/android-switch-backport)
- `DatePicker` (https://github.com/SimonVT/android-datepicker)
- `NumberPicker` (https://github.com/SimonVT/android-numberpicker)
- `TimePicker` (https://github.com/SimonVT/android-timepicker)

Avoid the New on the Old

If the goal is to support new capabilities on new devices, while not losing support for older devices, that implies we have the ability to determine what devices are newer and what devices are older. There are a few techniques for doing this, involving Java and resources.

Java

If you wish to conditionally execute some lines of code based on what version of Android the device is running, you can check the value of `Build.VERSION`, referring to the `android.os.Build` class. For example:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
  // do something only on API Level 9 and higher
}
```

Any device running an older version of Android will skip the statements inside this version guard and therefore will not execute.

That technique is sufficient for Android 2.0 and higher devices. If you are still supporting Android 1.x devices, the story gets a bit more complicated, and that will be discussed later in the book.

If you decide that you want your build target to match your `minSdkVersion` level — as some developers elect to do — your approach will differ. Rather than *blocking* some statements from being executed on *old* devices, you will *enable* some statements to be executed on *new* devices, where those statements use Java reflection (e.g., `Class.forName()`) to reference things that are newer than what your build target supports. Since using reflection is extremely tedious in Java, it is usually simpler to have your build target reflect the classes and methods you are actually using.

@TargetAPI

One problem with this technique is that your IDE will grumble at you, saying that you are using classes and methods not available on the API level you set for your `minSdkVersion`. To quiet down these Lint messages, you can use the `@TargetAPI` annotation.

For example, earlier in the book, we saw code like this:

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
static public <T> void executeAsyncTask(AsyncTask<T, ?, ?> task, T... params) {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, params);
  }
  else {
    task.execute(params);
```

```
  }
}
```

This utility method executes an `AsyncTask` using a multi-threaded thread pool. That is the default behavior of `execute()` on API Level 10 and below. On higher versions of Android, we can explicitly opt into the multi-threaded thread pool by using `executeOnExecutor()`, but that method does not exist prior to API Level 11. Hence, we check our API level at runtime via `Build.VERSION.SDK_INT`, see if we are on `HONEYCOMB` or higher, and branch accordingly. However, for a project with a `minSdkVersion` of 10 or below, Lint will still complain — Lint is just not sophisticated enough to realize that we are correctly handling newer API levels. The `@TargetApi(Build.VERSION_CODES.HONEYCOMB)` annotation tells Lint that we have indeed confirmed that we are "doing the right thing", at least through API Level 11.

However, by using `@TargetApi(Build.VERSION_CODES.HONEYCOMB)`, we are implicitly saying that we have *not* checked to see if we are doing things properly for higher versions of Android. So long as all the classes, methods, and such that we reference in this `executeAsyncTask()` method are available in API Level 11, we are fine. If we change the implementation to reference something from, say, API Level 14, now Lint will start complaining again. **This is what we want**, so we are alerted to the problem and can fix it. Hence, *only* set the `@TargetApi()` annotation to the API level that are you explicitly handling. Do not just set it to some arbitrarily high level (or, worse, use `@SuppressWarning` to try to get Lint to shut up entirely).

Resources

The aforementioned version guards only work for Java code. Sometimes, you will want to have different resources for different versions of Android. For example, you might want to make a custom style that inherits from `Theme.Holo` for Android 3.0 and higher. Since `Theme.Holo` does not exist on earlier versions of Android, trying to use a style that inherits from it will fail miserably on, say, an Android 2.2 device.

To handle this scenario, use the `-vNN` suffix to have two resource sets. One (e.g., `res/values-v11/`) would be restricted to certain Android versions and higher (e.g., API Level 11 and higher). The default resource set (e.g., `res/values/`) would be valid for any device. However, since Android chooses more specific matches first, an Ice Cream Sandwich phone would go with the resources containing the `-v11` suffix. So, in the `-v11` resource directories, you put the resources you want used on API Level 11 and higher, and put the backwards-compatible ones in the set without the suffix. This works for Android 2.0 and higher. You can also use `-v3` for resources that *only*

will be used on Android 1.5 (and no higher) or `-v4` for resources that *only* will be used on Android 1.6.

Components

One variation on the above trick allows you to conditionally enable or disable components, based on API level.

Every `<activity>`, `<receiver>`, or `<service>` in the manifest can support an `android:enabled` attribute. A disabled component (`android:enabled="false"`) cannot be started by anyone, including you.

We have already seen string resources be used in the manifest, for things like `android:label` attributes. Boolean values can also be created as resources. By convention, they are stored in a `bools.xml` file in `res/values/` or related resource sets. Just as `<string>` elements provide the definition of a string resource, `<bool>` elements provide the definition of a boolean resource. Just give the boolean resource a name and a value:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="on_honeycomb">false</bool>
</resources>
```

The above example has a boolean resource, named `on_honeycomb`, with a value of `false`. That would typically reside in `res/values/bools.xml`. However, you might also have a `res/values-v11/bools.xml` file, where you set `on_honeycomb` to `true`.

Now, you can use `@bool/on_honeycomb` in `android:enabled` to conditionally enable a component for API Level 11 or higher, leaving it disabled for older devices.

This can be a useful trick in cases where you might need multiple separate implementations of a component, based on API level. For example, later in the book we will examine app widgets — those interactive elements users can add to their home screens. App widgets have limited user interfaces, but API Level 11 added a few new capabilities that previously were unavailable, such as the ability to use `ListView`. However, the code for a `ListView`-backed app widget may be substantially different than for a replacement app widget that works on older devices. And, if you leave the `ListView` app widget enabled in the manifest, the user might try choosing it and crashing. So, you would only enable the `ListView` app widget on API Level 11 or higher, using the boolean resource trick.

Testing

Of course, you will want to make sure your app really does work on older devices as well as newer ones.

At build time, one trick to use periodically is to change your build target to match your `minSdkVersion`, then see where the compiler complains. If everything is known (e.g., resource attributes that will be ignored on older versions) or protected (e.g., Java statements inside a version guard `if` statement), then you are OK. If, however, you see complaints about something you forgot was only in newer Android releases, you can take steps to fix things.

You will also want to think about Android versions when it comes to testing, a topic that will be covered later in this book.

Keeping Track of Changes

Each Android SDK release is accompanied by API release notes, such as this set for Android 4.4/API Level 19.

Similarly, each Android SDK release is accompanied by its "API Differences Report", a roster of each added, removed, or modified class or method. For example, this API Differences Report points out the changes between API Level 18 and API Level 19.

Other changes are called out in the JavaDocs for `Build.VERSION_CODES`, with particular emphasis on what happens when you set a specific API level as your `android:targetSdkVersion`. Note that this roster is not complete, but may mention some things not mentioned in the other locations.

Each class, method, and field in the JavaDocs has a notation as to what API level that particular item was added. Class API levels appear towards the top of the page; method and field API levels appear on the right side of the gray bar containing the method signature or field declaration. Also, in the JavaDocs "Android APIs" column on the left, there is a drop-down that allows you to filter the contents based upon API level.

System Services

One of the problems that we have in Android app development is the overloading of terms. We have already seen how "layouts" sometimes refer to layout resources and sometimes refer to container classes like `LinearLayout`.

Another example comes in the name "service". This was already used in a few places in Java (e.g., `ExecutorService`). Android then used it for one of our four app components. Android *also* uses "service" as part of the term "system service"... where system services have little to do with Java services or Android services.

What is a System Service?

System services are "manager"-type classes that you get by calling `getSystemService()` on some `Context`, such as an `Activity` or `Service`. Usually, system services are tied to lower-level device functionality, like telephony. However, not all low-level device functionality is exposed by means of system services; some have separate APIs implemented by other sorts of "manager" classes.

There are two flavors of `getSystemService()`. The one that you are likely to use is the one that takes a `String` parameter that is the name of the system service that you want. You get back a generic `Object`, which you then have to downcast to the specific type of system service that you are trying to use:

```
AlarmManager mgr=(AlarmManager)someContext.getSystemService(Context.ALARM_SERVICE);
```

API Level 23 finally added a type-safe version of `getSystemService()`. You pass in the Java class object for the system service and get an instance of that class back:

```
AlarmManager mgr=someContext.getSystemService(AlarmManager.class);
```

However, until your `minSdkVersion` rises to 23 or higher, you will not be able to use that version of `getSystemService()` on `Context` on older devices.

Alas, there is no backport of `getSystemService()` on `ContextCompat` from the Android Support library.

What System Services Are There?

There are many system services, with new ones coming every Android version release or two. Here are the major ones as of Android 6.0, with links to chapters that focus on them (where available):

- `AccessibilityManager`, for being notified of key system events (e.g., activities starting) that might be relayed to users via haptic feedback, audio prompts, or other non-visual cues
- `AccountManager`, for working with Android's system of user accounts and synchronization
- `ActivityManager`, for getting more information about what processes and components are presently running on the device
- `AlarmManager`, for scheduled tasks (a.k.a., "cron jobs"), covered elsewhere in this book
- `AppOpsManager`, for "tracking application operations on the device"
- `AppWidgetManager`, for creating or hosting app widgets
- `AudioManager`, for managing audio stream volumes, audio ducking, and other system-wide audio affordances
- `BatteryManager`, for finding out about the state of the battery
- `BluetoothManager`, for exposing or connecting to Bluetooth services
- `ClipboardManager`, for working with the device clipboard, covered elsewhere in this book
- `ConnectivityManager`, for a high-level look as to what sort of network the device is connected to for data (e.g., WiFi, 3G)
- `ConsumerIrManager`, for creating "IR blaster" or other IR-sending apps, on hardware that has an IR transmitter
- `DevicePolicyManager`, for accessing device administration capabilities, such as wiping the device
- `DisplayManager`, for working with external displays, covered elsewhere in this book
- `DownloadManager`, for downloading large files on behalf of the user, covered in elsewhere in the book

- `DropBoxManager`, for maintaining your own ring buffers of logging information akin to Logcat
- `FingerprintManager`, for working with fingerprint readers on Android 6.0+ devices
- `InputMethodManager`, for working with input method editors
- `InputManager`, for identifying external sources of input, such as keyboards and trackpads
- `JobScheduler`, for scheduling periodic background work, covered elsewhere in the book
- `KeyguardManager`, for locking and unlocking the keyguard, where possible
- `LauncherApps`, for identifying launchable apps on the device (e.g., for home screen launchers), taking into account device policies
- `LayoutInflater`, for inflating layout XML files into Views, as you saw earlier in the book
- `LocationManager`, for determining the device's location (e.g., GPS), covered in the chapter on location tracking
- `MediaProjectionManager`, for capturing screenshots and screencasts
- `MediaRouter`, for working with external speakers and displays
- `MediaSessionManager`, for teaching Android about media that you are playing back
- `MidiManager`, for playing MIDI audio
- `NetworkStatsManager`, "for querying network usage stats"
- `NfcManager`, for reading NFC tags or pushing NFC content
- `NotificationManager`, for putting icons in the status bar and otherwise alerting users to things that have occurred asynchronously, covered in the chapter on `Notification`
- `NsdManager`, for network service discovery operations
- `PowerManager`, for obtaining `WakeLock` objects and such, covered elsewhere in this book
- `PrintManager`, for printing from Android
- `RestrictionsManager`, for identifying and working with restricted operations
- `SearchManager`, for interacting with the global search system
- `SensorManager`, for accessing data about sensors, such as the accelerometer, covered elsewhere in this book
- `StorageManager`, for working with expanded payloads (OBBs) delivered as part of your app's installation from the Play Store
- `SubscriptionManager`, for dealing with data roaming and other telephony subscription rules
- `TelecomManager`, for dealing with incoming and outgoing phone calls

- `TelephonyManager`, for finding out about the state of the phone and related data (e.g., SIM card details)
- `TextServicesManager`, for working with spelling checkers and other "text services"
- `TvInputManager`, for Android-powered televisions, to find out about TV inputs
- `UiModeManager`, for dealing with different "UI modes", such as being docked in a car or desk dock
- `UsageStatsManager`, "for querying device usage stats"
- `UsbManager`, for working directly with accessories and hosts over USB
- `UserManager`, for working with multiple user accounts on a compatible device (Android 4.2+ tablets, Android 5.0+ phones)
- `Vibrator`, for shaking the phone (e.g., haptic feedback)
- `WallpaperService`, for working with the device wallpaper
- `WifiManager`, for getting more details about the active or available WiFi networks
- `WifiP2pManager`, for setting up and communicating over WiFi peer-to-peer (P2P) networks
- `WindowManager`, mostly for accessing details about the default display for the device

Google Play Services

A term that you will encounter a fair bit as an Android developer is “Google Play Services”, or “Play Services” for short. This is your gateway into a series of proprietary capabilities that Google has layered on top of Android. Many of these capabilities are tied to Google’s servers and services, such as ads and Google Drive.

However, these capabilities, while usually free from monetary cost to the developer, are not free from problems or controversy.

What Is Google Play Services?

Google Play Services is a “kitchen sink” term, encompassing a wide range of things from the standpoint of developers and users alike.

…From the Standpoint of Developers?

The Play Services SDK allows you to integrate your Android app with a number of Google proprietary services, from leaderboard management for games to interacting with Chromecast devices. Many, but not all, of these services are tied to Google servers. Many, but not all, of these services will require some sort of API key as a result.

The SDK comes in the form of an Android library project that you link into your app, giving you access to classes and methods that let you add maps, or payment options, or push message receipt into your Android apps.

Note that while the name “Play Services” contains the word “services”, Play Services is merely an API, one that does not directly have anything to do with services or system services.

…From the Standpoint of Users of Google Play Devices?

In Western countries, the common perception is that all Android devices are part of the Google Play world. These devices will have the Play Services Framework pre-installed from the device manufacturer and silently updated over the air by Google. Apps that use the Play Services SDK in theory can use all of the SDK's available APIs on all devices equipped with the Play Services Framework.

In practice, older devices (particularly Android 2.x) will have some number of limitations related to Play Services, not the least of which being the lack of automatic over-the-air updates. As many developers are now setting their `minSdkVersion` to be something newer (e.g., 15), this particular class of problems will tend to fall by the wayside.

…From the Standpoint of the Android Ecosystem?

Google's continued expansion of the Play Services SDK, sometimes at the expense of Android itself, has not proven to be universally popular:

- Developers who depend on the Play Services SDK will not be able to run on devices that lack the Play Services Framework. And while many people think that the only devices that matter have the Play Services Framework, some estimates indicate that over half of Android devices in use today are from manufacturers that are not part of the Google Play ecosystem.
- The Play Services SDK is closed-source, and as such it makes debugging certain classes of problem more difficult.
- The terms and conditions for using different aspects of the Play Services SDK may cause problems for some developers, ranging from interfering with their planned business model to interfering with their planned software license (e.g., GPL).

What Is In the Play Services SDK?

As mentioned earlier, the Play Services SDK is *vast*. The following sub-sections outline some of the major pieces of the Play Services SDK, what Gradle dependency pulls them in, and what independent alternatives exist (if any).

Android Pay / Google Wallet

Google has tried a couple of times to get into the mobile payments market, starting with Google Wallet, which has now morphed into Android Pay. If you want to allow users to purchase goods and services through your app, and you want to allow those users to pay via Android Pay, you can use this portion of the Play Services SDK.

Android Wear

To communicate from a device running an open source operating system (Android, on a phone or tablet) to a device running an open source operating system (Android, on an Android Wear device), you have to use a proprietary, closed-source library.

It is possible to show a `Notification` on a Wear device straight from the Android SDK. It is also possible to create a Wear app that exists standalone straight from the Android SDK. But if you want to send data to the Wear device from the phone or tablet, or vice versa, that requires the Wear portion of the Play Services SDK.

This library provides a few discrete APIs for communication:

- A shared data API, where both sides can read and write from a key-value store that is synchronized between the two environments
- A message API, for a classic point-to-point communications pattern
- An asset transfer API, designed for larger data sets (e.g., large images)

Google+

The documentation and business proposition for the Google+ API is a bit limited at this time. However, it appears that you can:

- add a +1 button to your app, if that sounds interesting
- have richer options for sharing content to a user's Google+ account, beyond simple `ACTION_SEND`
- examine the user's Google+ profile and some of the user's friends on Google+

Google Account Login / Sign In with Google

Rather than maintain your own account system, your app could ask the users to sign into their Google account as part of using your app.

Google Analytics

"Analytics" refers to tracking usage. Web analytics uses a mix of Web server logs, tracking cookies, and the like to determine popular Web pages, navigation flows, time spent in certain areas of a site, and so forth. Mobile analytics tracks usage within an app: certain activities, certain operations, etc.

Google Analytics is very popular for Web sites, and Google extended this to a mobile API designed for tracking app usage.

There are countless analytics services with Android APIs (e.g., Flurry) beyond Google's. While there appear to be few self-hosted or open source solutions, analytics data collection is not especially difficult to implement on your own, if you would prefer to keep this information more private. Data analysis is where the challenges with home-grown solutions arise. Or, you could simply not collect this sort of information.

Google App Indexing

Google App Indexing, among other things, allows for "deep links" into an Android app, surfaced from Google search results. That, on its own, does not require any particular proprietary APIs. However, to allow Google to discover these "deep links", it appears that you need to use a custom app-indexing API.

Google App Invites

Google's App Invites service allows your users to annoy their contacts, bugging them to install your app.

A simpler, albeit less slick, solution is to allow the user to send messages from your app with a link back to your app from its distribution channel (e.g., Play Store), such as via an `ACTION_SEND` `Intent`.

Google Cast

Google Cast can be thought of as a control protocol for Google Cast-enabled receivers. Through a Google-supplied SDK (or other means), Google Cast client apps ("senders") can direct a Google Cast-enabled receiver to play, pause, rewind, fast-forward, etc. a stream. Android TV devices and Chromecast devices are the primary Cast-enabled receivers.

Google Cast does assume that, in general, the media receiver runs its own OS and is capable of playing streaming media without ongoing assistance from the Google Cast client. Hence, the client is not "locked into" having to keep feeding content to the Google Cast client, allowing the user to go off and do other things with that client while playback is going on.

Chromecast offers up remote playback media routes and works with `RemotePlaybackClient`, as is discussed in the chapter on `MediaRouter`. The sample app for `RemotePlaybackClient` was tested on a Chromecast.

If you want greater control than is offered via `RemotePlaybackClient`, though, you can use the Cast SDK. However, using the Cast SDK will tie you to Google Cast — and some of its restrictions, both technical and legal — but will give you greater developer control over the behavior of both the Google Cast device and your app.

As noted above, `RemotePlaybackClient`, along with the `Presentation` API, offer a significant subset of what the Cast SDK offers.

Google Cloud Messaging

Google Cloud Messaging – GCM for short — asynchronously delivers notifications from the Internet ("cloud") to Android devices. Rather than the device waking up and polling on a regular basis at the behest of your app, your app can register for notifications and then wait for them to arrive. GCM is engineered with efficiency in mind:

- Apps do not have to be constantly running, maintaining their own socket connections to some XMPP or MQTT server (let alone several such apps)
- Apps can share a single managed connection to a Google server, one that is carefully tuned to minimize power draw while also keeping the connection alive
- Apps can avoid frequent wakeup events for polling, letting some server do the "heavy lifting" and just tap the app on the virtual shoulder to inform it of some data of interest

The proper use of GCM means better battery life for your users. It can also reduce the amount of time your code runs, which helps you stay out of sight of users looking to pounce on background tasks and eradicate them with task killers.

GCM has gone through four revisions of its API, including the 2016 rebranding of it as Firebase Cloud Messaging (FCM). Be sure to use up-to-date references and examples when adding GCM/FCM to your apps.

You may also encounter references to "C2DM", GCM's precursor. C2DM debuted in 2010 and quickly became popular, for everything from triggering near-real-time data synchronization (e.g., Remember the Milk to-do list updates) to lightweight coordination between multiple players in a game. However, C2DM was a Google Labs product and in perpetual beta form. When Google Labs was shut down, C2DM was in limbo: not canceled, but not converted into an actual product. In 2012, GCM formally replaced C2DM, and in 2015, C2DM was shut down entirely. Hence, while high-level concepts about push messaging from the C2DM era might still be relevant to you, any actual C2DM-related code will be useless.

Other devices from outside the Google Play ecosystem may offer their own counterparts to GCM. Independent push implementations can range from XMPP and MQTT to simple WebSockets, though these have limitations when compared to GCM.

Google Drive

Google Drive is Google's hosted file-storage service. Via Drive APIs in the Play Services SDK, you can work indirectly with the user's Google Drive-hosted content, including creating and deleting files, plus searching through files for ones that meet particular search criteria.

Note that some of this functionality is available via the Storage Access Framework in Android 4.4+, with the advantage that it works across multiple content sources, not just Google Drive.

Other services (e.g., Dropbox) have their own APIs as well.

Google Fit

Google Fit is Google's wearable sensor initiative, for "smartbands" and related gadgets. Through the Fit APIs, you can detect Fit gadgets associated with a user's device, read data from those gadgets' sensors (e.g., heart rate), and so forth.

Other manufacturers in this space (e.g., Fitbit) have their own SDKs as well.

Google Location Services

This portion of the Play Services SDK offers the "fused location provider". This combines GPS and network sources of location data, plus sensor information, to try to offer better location information with less power draw. For example, if the sensors suggest that the device is not moving, the fused location provider can scale back how aggressively it uses the location sources, since the location probably is not changing.

This library also offers a "geofencing" implementation, where you ask the Play Services SDK to keep track of certain locations and let you know if the device gets within a certain distance of those locations.

This book has a chapter on the fused location provider.

Google Maps

Android has offered integrated Google Maps to developers since the outset. With the introduction of Maps V2 in 2012, this capability was folded into the Play Services SDK. Through Maps V2, you can embed a map powered by Google Maps into your application, complete with markers and popups, lines and shaded areas, and so on.

This book has a chapter on Maps V2.

Due to the popularity of embedded maps, other manufacturers (e.g., Amazon, Blackberry) have offered their own map engines, often with APIs that attempt to mimic that of Maps V2 (or perhaps its predecessor, now known as Maps V1). Beyond that, there is the OpenStreetMap project, for which Android libraries are available.

Google Mobile Ads / AdMob

Google is an advertising company. They offer the Google Mobile Ads SDK (a.k.a., AdMob for Android) as part of the Play Services SDK, for you to be able to add banners, interstitials, and other forms of advertising to your app.

There are other competing mobile ad networks that you could consider, though you may be better served focusing on coming up with a better business model.

Google Mobile Vision

Google has a variety of APIs, grouped under the "Mobile Vision" banner, designed for detecting specific sorts of objects or other information in still photos and videos. These include:

- detection of faces, and the state of those faces (e.g., expressions)
- detection and decoding of barcodes

Android's native camera API has some amount of face recognition, though not to the level of the Face API in the Mobile Vision SDK.

There are a variety of barcode scanning apps (e.g., the legendary ZXing Barcode Scanner) and libraries (e.g., ZBar) that one can use independently of the Play Services SDK.

Google Nearby

Google Nearby offers a pair of APIs for communication between nearby devices.

The Nearby Messages API offers a publish-and-subscribe messaging framework, designed for sending small blocks of data between Internet-connected Android and iOS devices. This is largely frictionless for the user (beyond the network connection), as the Messages API uses a mix of radios (Bluetooth, Bluetooth LE, WiFi) and ultrasonic signaling to handle the pairing and interaction.

The Nearby Connections API offers connection-based group messaging between devices on the same WiFi network. While you can pass more data this way, since everybody has to be on WiFi, it reduces the number of potential communications partners.

While some aspects of Google Nearby (e.g., ultrasound) are unusual, there have been many projects offering server-less group communications, from ZeroMQ to AllJoyn.

SafetyNet

The SafetyNet APIs lets your app know "whether the device where it is running matches the profile of a device that has passed Android compatibility testing". Presumably, this is designed to help you detect custom ROMs or copies of your app installed from pirate sites onto incompatible hardware.

Adding Play Services to Your Project

On the surface, using Play Services should be simple: add the aforementioned `implementation` statement(s), then start calling some methods from the supplied Play Services SDK libraries.

Unfortunately, it is not that simple. There are a number of other things that you will need to deal with in order to integrate Play Services into your app.

The Metadata

You will see plenty of examples that show having a `<meta-data>` element, inside your `<application>` element, with an `android:name` of `com.google.android.gms.version` and a value pulling in an integer resource (`@integer/google_play_services_version`) from the Play Services SDK:

```
<application
  android:allowBackup="false"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/Theme.Apptheme">
  <activity android:name=".WeatherDemo">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>

      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
  <activity android:name=".LegalNoticesActivity"/>

  <meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version"/>
</application>
```

(from Location/FusedNew/app/src/main/AndroidManifest.xml)

This is no longer required, if you are using version 8.1.0 or higher of the Play Services SDK. This element will be added to your manifest automatically via the manifest merger process.

Dealing with Runtime Permissions

Android 6.0's runtime permission system affects some of the Play Services APIs. For example, if you are trying to get the location via the fused location provider, you will need `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. Both of those are `dangerous` permissions, so apps with a `targetSdkVersion` of 23 or higher will need to request those permissions at runtime.

The `Location/FusedNew` sample application contains an `AbstractGoogleApiClientActivity` that, among other things, helps us deal with runtime permissions in our Play Services SDK-using apps.

Detecting If We Have Permission

The idea behind `AbstractGoogleApiClientActivity` is that apps using the Play Services SDK will have activities that inherit from `AbstractGoogleApiClientActivity`, overriding a few methods to configure how `AbstractGoogleApiClientActivity` handles things like runtime permissions. For example, `AbstractGoogleApiClientActivity` has an `abstract` method named `getDesiredPermissions()` that subclasses must override, providing a `String` array of permissions that the activity needs. `AbstractGoogleApiClientActivity` then uses `hasAllPermissions()` and `hasPermission()` private methods to determine whether all of the requested permissions are currently held:

```
private boolean hasAllPermissions(String[] perms) {
  for (String perm : perms) {
    if (!hasPermission(perm)) {
      return(false);
    }
  }

  return(true);
}

private boolean hasPermission(String perm) {
  return(ContextCompat.checkSelfPermission(this, perm)==
    PackageManager.PERMISSION_GRANTED);
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

In `onCreate()` of `AbstractGoogleApiClientActivity`, among other things, we call `hasAllPermissions()` to see if we have all of our required permissions — if yes, we

can go ahead and call an `initPlayServices()` method to start the process of initializing our access to the Play Services SDK:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (savedInstanceState!=null) {
    isInPermission=
      savedInstanceState.getBoolean(STATE_IN_PERMISSION, false);
    isResolvingPlayServicesError=
      savedInstanceState.getBoolean(STATE_IN_RESOLUTION, false);
  }

  if (hasAllPermissions(getDesiredPermissions())) {
    initPlayServices();
  }
  else if (!isInPermission) {
    isInPermission=true;

    ActivityCompat
      .requestPermissions(this,
        netPermissions(getDesiredPermissions()),
        REQUEST_PERMISSION);
  }
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

Requesting Permissions

If we do not have all of the permissions, `onCreate()` will call `requestPermissions()` on `ActivityCompat` to ask the user for them. However, it also leverages `netPermissions()` to filter out the permissions that the user previously granted, so we only bother the user with permissions that either the user has not seen before or has previously denied:

```
private String[] netPermissions(String[] wanted) {
  ArrayList<String> result=new ArrayList<String>();

  for (String perm : wanted) {
    if (!hasPermission(perm)) {
      result.add(perm);
    }
  }
```

```
    return(result.toArray(new String[result.size()]));
  }
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

Note that this code is not making use of `shouldShowRequestPermissionRationale()`, to detect previous permission denials and perhaps show some UI to educate the user on what the impacts are of this rejection.

Handling the Result

The call to `requestPermissions()` will eventually trigger a callback to `onRequestPermissionsResult()`. Here, if we now have all of the permissions, we call `initPlayServices()` (more on this in a bit) and then `connect()` to the Play Services SDK (also, more on this in a bit):

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  isInPermission=false;

  if (requestCode==REQUEST_PERMISSION) {
    if (hasAllPermissions(getDesiredPermissions())) {
      initPlayServices();
      playServices.connect();
    }
    else {
      handlePermissionDenied();
    }
  }
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

If, however, we do not have all of the requested permissions, another `abstract` method on this class is `handlePermissionDenied()`, where the subclass can do what it wants to. That could range from explaining to the user what can and cannot be done to simply calling `finish()` and going away.

Dealing with Configuration Changes

There is a possibility that the user will rotate the screen or otherwise trigger a configuration change while we are in the request-permission process. Even though our activity is not in the foreground from an input standpoint, it *is* visible, and so it will undergo the configuration change while the request-permission dialog is still in the foreground. We do not want to pop up the dialog *again* (and confuse the user). So, the `isInPermission` field is tracking whether the request-permission dialog is outstanding, so we do not attempt to show the dialog again in `onCreate()`.

Since the activity could be destroyed and recreated as part of the configuration change, we hang onto the `isInPermission` value in the saved instance state `Bundle`:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putBoolean(STATE_IN_PERMISSION, isInPermission);
  outState.putBoolean(STATE_IN_RESOLUTION,
    isResolvingPlayServicesError);
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

(the `STATE_IN_RESOLUTION` bit will be explained shortly)

And, in `onCreate()`, we re-initialize `isInPermission` if we got the saved instance state `Bundle` passed in.

Checking for Play Services

While you typically think of Android devices as having Play Services, that is not always the case. Sometimes, they do not, yet wind up having a copy of your app anyway, perhaps through less-than-legal measures. Or, the device has Play Services, but it is not the latest version — perhaps the user missed a recent Play Services update due to international travel, taking up temporary residence in a Faraday cage, or other technical issues.

Hence, another thing that `AbstractGoogleApiClientActivity` does is confirm that the device has Play Services and can connect to the Play Services process for whatever particular API(s) we wish to use.

Initializing the GoogleApiClient

For many, though not all, Play Services APIs, you use a `GoogleApiClient` as your entry point for talking to Play Services. Some APIs, like Maps V2, do not use `GoogleApiClient` for some reason. But, more often than not, you will find yourself needing `GoogleApiClient`.

To create a `GoogleApiClient` instance, use a `GoogleApiClient.Builder`. As the class name suggests, `GoogleApiClient` is used as a client connection to many (but not all) Google Play Services APIs, and `GoogleApiClient.Builder` is a builder for building such a connection. In particular:

- We pass our `Activity` as our `Context`
- We call `addConnectionCallbacks()` to indicate what should be notified when our connection to the Play Services process is ready
- We call `addOnConnectionFailedListener()` to indicate what should be notified if we have a problem connecting to the Play Services process
- We call `build()` on the `Builder` to actually build the `GoogleApiClient`

This is handled by `initPlayServices()` on `AbstractGoogleApiClientActivity`, which we call once we have our permissions set up:

```
protected void initPlayServices() {
  playServices=
    configureApiClientBuilder(new GoogleApiClient.Builder(this))
      .addConnectionCallbacks(this)
      .addOnConnectionFailedListener(this)
      .build();
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

This includes calling out to the subclass' implementation of `configureApiClientBuilder()`, where the subclass can use methods like `addApi()` to indicate specifically what parts of the Play Services family of APIs the activity wants to use.

Given that we have a `GoogleApiClient`, we need to `connect()` to it to be able to start requesting location data, then `disconnect()` from it when we no longer need that location data.

Disconnecting is easy: we do that in `onStop()` of `AbstractGoogleApiClientActivity`:

```
@Override
protected void onStop() {
  if (playServices!=null) {
    playServices.disconnect();
  }

  super.onStop();
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

There are two places where we possibly call `connect()`. One is if we needed to ask for permissions, and the user granted them. In `onRequestPermissionsResult()`, after confirming that we do indeed have all necessary permissions, we call `initPlayServices()` and then immediately call `connect()` on the `GoogleApiClient`:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  isInPermission=false;

  if (requestCode==REQUEST_PERMISSION) {
    if (hasAllPermissions(getDesiredPermissions())) {
      initPlayServices();
      playServices.connect();
    }
    else {
      handlePermissionDenied();
    }
  }
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

If we did *not* need to request permissions, we call `connect()` in `onStart()`, mirroring the `onStop()` where we are disconnecting:

```
@Override
protected void onStart() {
  super.onStart();

  if (!isResolvingPlayServicesError && playServices!=null) {
    playServices.connect();
```

```
    }
  }
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

The `isResolvingPlayServicesError` `boolean` value will be discussed a bit later in this chapter.

Connecting and Disconnecting

The call to `connect()`, in turn, will trigger calls to our `onConnected()` and `onDisconnected()` methods of the `GoogleApiClient.ConnectionCallbacks` interface, assuming all goes well. `AbstractGoogleApiClientActivity` does not provide those implementations; they are considered part of the `abstract` API and therefore need to be implemented by subclasses.

However, apparently it is possible for this connection attempt to fail. Exactly how and why it might fail is not well documented. If it fails, the `onConnectionFailed()` method from our `GoogleApiClient.OnConnectionFailedListener` implementation will be called. `onConnectionFailed()` is passed a `ConnectionResult` indicating what specifically went wrong.

It turns out that this `ConnectionResult` may contain a `PendingIntent` that can be used to try to help the user recover from whatever the problem was. The recipe that we have been given to try to use this is to call `hasResolution()` (to see if the `PendingIntent` exists) and to use `startResolutionForResult()` (to invoke the activity pointed to by the `PendingIntent`). Of course, `hasResolution()` may return `false`, and apparently the `PendingIntent` might be broken, so we have to handle those scenarios as well:

```
@Override
public void onConnectionFailed(ConnectionResult result) {
  if (!isResolvingPlayServicesError) {
    if (result.hasResolution()) {
      try {
        isResolvingPlayServicesError=true;
        result.startResolutionForResult(this, REQUEST_RESOLUTION);
      }
      catch (IntentSender.SendIntentException e) {
        playServices.connect();
      }
    }
    else {
      ErrorDialogFragment.newInstance(result.getErrorCode())
```

```
          .show(getFragmentManager(),
            TAG_ERROR_DIALOG_FRAGMENT);
        isResolvingPlayServicesError=true;
      }
    }
  }
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

If we have a resolution and successfully start up the resolution activity, our activity will be stopped and later started, at which point we will wind up trying to `connect()` again naturally.

If there is no `PendingIntent` to try to resolve the problem, we can still attempt to display a dialog with information about what is going wrong. The Play Services SDK provides this dialog, though we are responsible for wrapping it in a `DialogFragment` ourselves. That comes in the form of `ErrorDialogFragment`:

```
public static class ErrorDialogFragment extends
  DialogFragment {
  static final String ARG_ERROR_CODE="errorCode";

  static ErrorDialogFragment newInstance(int errorCode) {
    Bundle args=new Bundle();
    ErrorDialogFragment result=new ErrorDialogFragment();

    args.putInt(ARG_ERROR_CODE, errorCode);
    result.setArguments(args);

    return(result);
  }

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    return(GoogleApiAvailability
      .getInstance()
      .getErrorDialog(
        getActivity(),
        getArguments().getInt(ARG_ERROR_CODE),
              REQUEST_RESOLUTION));
  }

  @Override
  public void onCancel(DialogInterface dlg) {
    if (getActivity()!=null) {
      getActivity().finish();
    }
```

```
    super.onCancel(dlg);
  }

  @Override
  public void onDismiss(DialogInterface dlg) {
    if (getActivity()!=null) {
      ((AbstractGoogleApiClientActivity)getActivity())
        .isResolvingPlayServicesError=false;
    }

    super.onDismiss(dlg);
  }
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/AbstractGoogleApiClientActivity.java)

`onCreateDialog()` uses the `GoogleApiAvailability` singleton to show the error dialog, given the error code that came from our previous attempt to connect. We pass that error code over to the `ErrorDialogFragment` via the arguments `Bundle`, so that it can survive a configuration change.

However, we also have to take into account that the device might undergo a configuration change while either the resolution activity started by `startActivityForResult()` or the `ErrorFragmentDialog` is in the foreground. What we do not want to do is immediately try connecting to Play Services again in `onStart()`, while we are in the process of trying to fix whatever problem prevented us from connecting to it previously.

So, we have to track a `boolean` state, `isResolvingPlayServicesError`, as a field in our activity. That is initially set to `false`, but we `flip` it to `true` if we show the resolution activity or the `ErrorFragmentDialog`. We flip it back to `false` when either the started activity returns control to us in `onActivityResult()` or when the `ErrorDialogFragment` is dismissed. While that flag is `true`, we skip attempting to connect to Play Services in `onStart()`. And this flag is part of our saved instance state, so we can handle configuration changes.

Getting Help

Obviously, this book does not cover everything. And while your #1 resource (besides the book) is going to be the Android SDK documentation, you are likely to need information beyond what's covered in either of those places.

Searching online for "android" and a class name is a good way to turn up tutorials that reference a given Android class. However, be sure to check the age of the blog post or whatever that you are reading. The older it is, the more likely that it is out of date, based upon changes in Android or just better solutions that have evolved over time.

Beyond randomly hunting around for tutorials, though, this chapter outlines some other resources to keep in mind.

Questions. Sometimes, With Answers.

Stack Overflow's android tag is the world's #1 place to get Android help. However, there are a variety of other sites offering help in a variety of languages.

It is important, particularly for Stack Overflow, to write well-written questions:

1. Include relevant portions of the source code (e.g., the method in which you are getting an exception) and the stack trace from Logcat, if the problem is an unhandled exception.
2. On Stack Overflow, make sure your source code and stack trace are formatted as source code; on Google Groups, consider posting long listings on gist.github.com or a similar sort of code-paste site.

3. Explain thoroughly what you are trying to do, how you are trying to do it, and why you are doing it this way (if you think your goal or approach may be a little offbeat).
4. On Stack Overflow, respond to answers and comments with your own comments, addressing the person using the @ syntax (e.g., `@CommonsWare`), to maximize the odds you will get a reply. However, only use that for people who are already involved in your question.
5. On the Google Groups, do not "ping" or reply to your own message to try to elicit a response until a reasonable amount of time has gone by (e.g., 24 hours).

Heading to the Source

The source code to Android is now available. Mostly this is for people looking to enhance, improve, or otherwise fuss with the insides of the Android operating system. But, it is possible that you will find the answers you seek in that code, particularly if you want to see how some built-in Android component "does its thing".

The source code and related resources can be found at http://source.android.com. Here, you can:

1. Download the source code
2. File bug reports against the operating system itself
3. Submit patches and learn about the process for how such patches get evaluated and approved
4. Join a separate set of Google Groups for Android platform development

Trail: Code Organization and Gradle

Working with Library Modules

Android library modules are the primary unit of Android source reuse, particularly where that source involves more than just Java source code, such as Android resources.

In this chapter, we will explore the basics of setting up and using an Android library module.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Creating a Library Module

An Android library module, in many respects, looks like a regular Android project. It has source code and resources. It has a manifest.

What it does not do, though, is build an APK file. Instead, it represents a basket of programming assets that the Android build tools know how to blend in with regular Android projects.

Making a project be an Android library module is simply a matter of choosing the right Android Gradle Plugin.

Rather than have:

```
apply plugin: 'com.android.application'
```

use:

```
apply plugin: 'com.android.library'
```

That's it — the `com.android.library` plugin now knows that it is creating a library, not an app.

The real question is, *where* are you making this library? In many cases, you will do so as a module in a project, where there is another module that is an app. This covers both:

- A library designed to be standalone, but with a sample app demonstrating its use
- A library designed to be used by the app, and perhaps other app modules in this project

Adding new modules to an Android Studio project is handled most simply via the new-module wizard, which you can bring up via File > New > New Module... from the main menu. This brings up the first page of the new-module wizard:

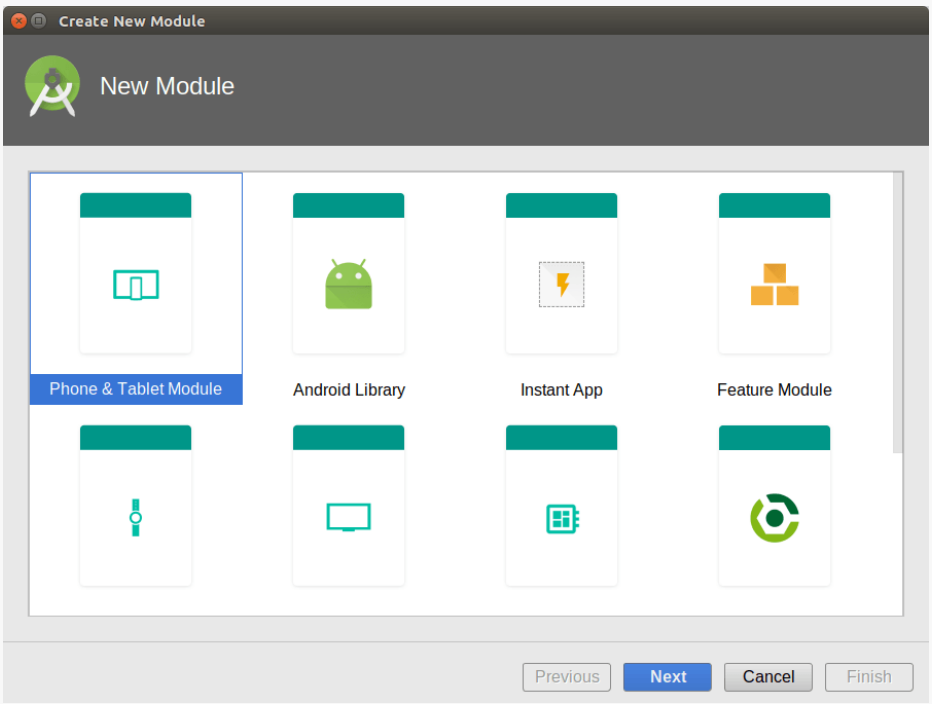


Figure 330: Android Studio New-Module Wizard, First Page

To add a library module as a module to an existing project, choose "Android Library" in the list of module types, then click Next to proceed to the second page of the wizard:

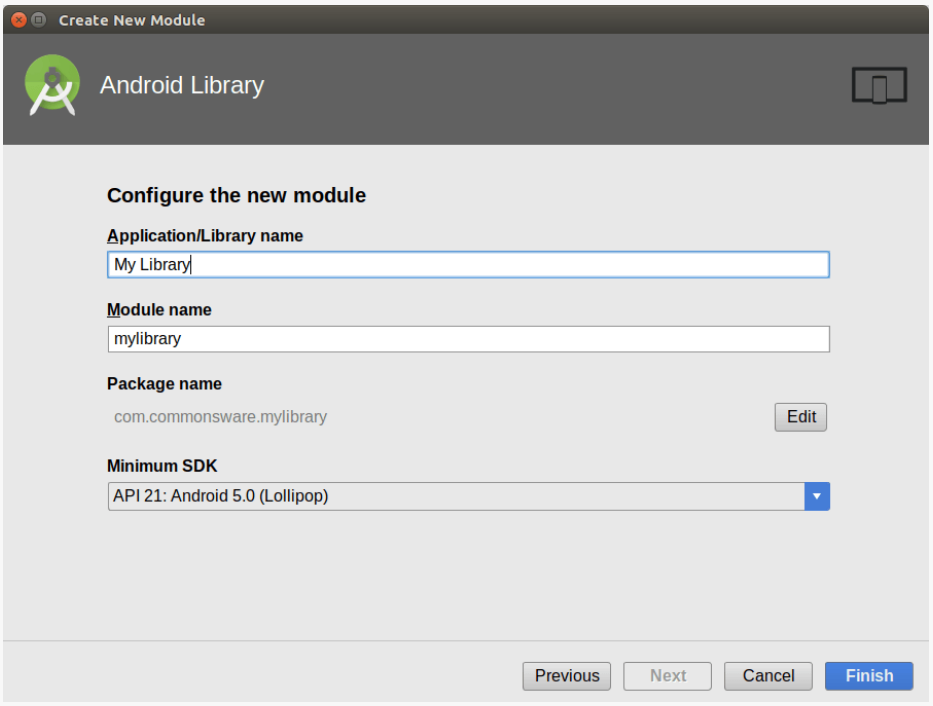


Figure 331: Android Studio New-Module Wizard, Second Page

This collects some bits of information, including:

- the "application name", whose use in this case is unclear
- the "module name", which will be the directory in this project into which this library will be created
- the "package name", which will go into the manifest of the generated library
- your module's `minSdkVersion`

At this point, clicking Next will take you to the same new-activity flow that you saw when creating a new project. If you want an activity to be generated for you in this library, proceed by selecting the activity template and providing the activity template configuration data. If you do not want an activity, choose "Add No Activity" in the grid of templates, then click "Finish" to create the module.

In the end, the new-module wizard will set up the new module for you, in your designated subdirectory of the project, including modifying `settings.gradle` to list

this subdirectory as being a module within the project. At this point, you will be able to start using the library within the project itself.

Using a Library Module

Once you have a library module, you can attach it to another Android module, so the other Android module has access to everything in the library. This works similarly to adding other sorts of dependencies, with a slight syntax change. Rather than use an artifact identifier or `fileTree()`, we use `project()`:

```
implementation project(':libname')
```

where `libname` is the name of the module. This will be the directory name, and it will also show up as the module's name in `settings.gradle` in the project root directory:

```
include ':app', ':libname'
```

The leading colon means that this module is directly off of the project root directory.

If you wish to use this library module in separate projects, other than the one that hosts the module, you will need to distribute it to those other projects. A typical way to do that is to compile the library module into an AAR file and have that be hosted in an artifact repository.

Library Modules and the Manifest

Library projects can publish their own `AndroidManifest.xml` file, which contributes to the overall manifest used by apps that incorporate the library. Hence, a library can:

- request permissions that perhaps are not in the app's own manifest
- publish activities or other components, without the app developer having to add entries to the app's own manifest
- stipulate a minimum SDK version required by the library code, which might be higher than the minimum SDK version required by the app itself

However, merging these manifests is a rather complex topic, and as such will be covered much later in the book.

Library Modules and Transitive Dependencies

Just as an application module can have dependencies, so can a library module. For other modules that depend upon the library module, the library module's dependencies become transitive dependencies. In other words, if App A depends upon Library B, and Library B depends upon Library C and Library D, App A has transitive dependencies on Library C and Library D.

When a module depends directly upon a library module (e.g., `implementation project(':libname')`), all of that library module's dependencies get added to the requesting module's dependencies list. The same holds true when a module depends upon a library from an artifact repository. It does *not* hold true for bare libraries, such as adding plain JARs to a project via `implementation fileTree(...)`.

There are three key ways that a library module can declare a dependency: `implementation`, `api`, and `compileOnly`. While an app module typically would only use `implementation`, all three of these options are possible for a library module, with subtly different behavior:

Configuration Name	Is Dependency's API Available to Hosts?	Is Dependency's Code Added to Hosts?
`implementation`	no	yes
`api`	yes	yes
`compileOnly`	no	no

Let's examine a concrete example: CWAC-NetSecurity. This library — which will be profiled later in the book — has its own code, plus it has helper code for integrating the library's code with OkHttp. There are three ways that CWAC-NetSecurity could declare its dependency upon OkHttp, and those choices change what happens in an app that itself depends upon CWAC-NetSecurity:

- If CWAC-NetSecurity uses `implementation` to depend upon OkHttp, then an app that depends upon CWAC-NetSecurity gets both CWAC-NetSecurity and OkHttp code added to the app ("Is Dependency's Code Added to Hosts?"). However, the OkHttp code itself is *not* deemed to be part of the API of CWAC-NetSecurity, and so the app that depends upon CWAC-NetSecurity cannot use the OkHttp classes, unless it has its own separate `implementation` dependency upon OkHttp.
- If CWAC-NetSecurity uses `api` to depend upon OkHttp, then as with `implementation`, both the code from OkHttp and the code from CWAC-

NetSecurity get added to apps that depend upon CWAC-NetSecurity. However, in this case, the app can directly refer to classes from either CWAC-NetSecurity or OkHttp, without an additional dependency declaration. In effect, CWAC-NetSecurity is stating that OkHttp is part of CWAC-NetSecurity's public API.
- What *really* happens is that CWAC-NetSecurity uses `compileOnly`. This says that OkHttp is used only to be able to compile CWAC-NetSecurity's own code. The OkHttp code is *not* added to apps that depend upon CWAC-NetSecurity, unless those apps have their own `implementation` dependency upon OkHttp.

`implementation` and `api` are the two most common. `compileOnly` is mostly for cases where the library module would like to offer integration for some third-party library, but where that library is not essential. For apps that want to use the integration, they simply need to have an additional dependency. For apps that *do not* want the integration, though, they avoid having all the extra code added from the third-party library.

Limitations of Library Modules

While library modules are useful for code organization and reuse, they do have their limits.

As noted above, if more than one project (main plus libraries) defines the same resource, the higher-priority project's copy gets used. Generally, that is a good thing, as it means that the main project can replace resources defined by a library (e.g., change icons). However, it does mean that two libraries might collide. It is important to keep your resource names distinct to minimize the odds of this occurrence.

Gradle and Tasks

A `build.gradle` file teaches Gradle how to execute tasks, such as how to compile an Android project. Outside of a Gradle-aware IDE like Android Studio, you use Gradle itself to run these tasks. If you have installed your own copy of Gradle, you would use the `gradle` command; if you are relying upon a trusted copy of the Gradle Wrapper, you would use the `./gradlew` script in your project root.

For the purposes of this book, the `gradle` command will be shown – just substitute `./gradlew` where you see `gradle` if you are using the Gradle Wrapper script.

Key Build-Related Tasks

To find out what tasks are available to you, you can run `gradle tasks` from the project directory. That will result in output akin to:

```
> Task :tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Android tasks
-------------
androidDependencies - Displays the Android dependencies of the project.
signingReport - Displays the signing info for each variant.
sourceSets - Prints out all the source sets defined in this project.

Build tasks
-----------
assemble - Assembles all variants of all applications and secondary packages.
assembleAndroidTest - Assembles all the Test applications.
```

```
assembleDebug - Assembles all Debug builds.
assembleRelease - Assembles all Release builds.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
clean - Deletes the build directory.
cleanBuildCache - Deletes the build cache directory.
compileDebugAndroidTestSources
compileDebugSources
compileDebugUnitTestSources
compileReleaseSources
compileReleaseUnitTestSources
mockableAndroidJar - Creates a version of android.jar that's suitable for unit tests.

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root project
'MyApplication'.
components - Displays the components produced by root project 'MyApplication'.
[incubating]
dependencies - Displays all dependencies declared in root project 'MyApplication'.
dependencyInsight - Displays the insight into a specific dependency in root project
'MyApplication'.
dependentComponents - Displays the dependent components of components in root project
'MyApplication'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'MyApplication'. [incubating]
projects - Displays the sub-projects of root project 'MyApplication'.
properties - Displays the properties of root project 'MyApplication'.
tasks - Displays the tasks runnable from root project 'MyApplication' (some of the
displayed tasks may belong to subprojects).

Install tasks
-------------
installDebug - Installs the Debug build.
installDebugAndroidTest - Installs the android (on device) tests for the Debug build.
uninstallAll - Uninstall all applications.
uninstallDebug - Uninstalls the Debug build.
uninstallDebugAndroidTest - Uninstalls the android (on device) tests for the Debug
build.
uninstallRelease - Uninstalls the Release build.

Verification tasks
```

```
------------------
check - Runs all checks.
connectedAndroidTest - Installs and runs instrumentation tests for all flavors on
connected devices.
connectedCheck - Runs all device checks on currently connected devices.
connectedDebugAndroidTest - Installs and runs the tests for debug on connected
devices.
deviceAndroidTest - Installs and runs instrumentation tests using all Device
Providers.
deviceCheck - Runs all device checks using Device Providers and Test Servers.
lint - Runs lint on all variants.
lintDebug - Runs lint on the Debug build.
lintRelease - Runs lint on the Release build.
lintVitalRelease - Runs lint on just the fatal issues in the release build.
test - Run unit tests for all variants.
testDebugUnitTest - Run unit tests for the debug build.
testReleaseUnitTest - Run unit tests for the release build.
```

This list is dynamically generated based on the contents of `build.gradle`, notably including tasks defined by the `com.android.application` plugin.

In principle, you are supposed to specify the entire task name when running that task. However, you can use shorthand, so long as it uniquely identifies the task.

Probably the most common task that a developer will use, at least in the short term, is `installDebug` (or `iD` for short). This will build a debug version of the app and install it on an available device or emulator. This roughly corresponds to `ant install debug` for those familiar with legacy Ant-based command-line builds.

Just as there is `installDebug`, there can also be `installRelease`. The `Debug` and `Release` portions of the task are not hard-coded, but rather are derived from the "build types" defined in the `build.gradle` file. The concept, role, and usage of build types will be covered in the next chapter. However, `installRelease` is not available by default, because installing an app requires that the APK be signed, and the Android Gradle Plugin does not know how to sign it. We will address this in the next chapter as well.

If you just want to build the app, without installing it, `assembleDebug` (`aD`) or `assembleRelease` (`aR`) will accomplish that aim. If you want to uninstall the app from a device or emulator, `uninstallDebug` (`uD`) and `uninstallRelease` (`uR`) should work.

Discussion of other tasks, such as the "check" tasks, will be covered in later chapters.

Results

All build output goes into a `build/` directory. Specifically, your APKs will go into `build/outputs/apk`, with different APK editions based upon whether you did a debug or release build.

Note that Gradle has a `clean` task that wipes out the `build/` directory.

Gradle Build Variants

You may think that the directory structure and files involved in an Android Studio project is a bit complicated. In truth, what you have seen so far is actually fairly simple, when you start to consider the real world of product development. Simple apps can get by with simple structures like we have seen so far, but:

- Many development teams want different behavior for debug builds instead of production builds, such as hitting a test server rather than the real app server
- Some development teams have different variations on the same app, tailored to different audiences, such as for distribution through different channels

In this chapter, we will explore the concept of "build variants" in Gradle and Android Studio and how you can take advantage of them for your projects.

Prerequisites

Understanding this chapter requires that you have read the chapters the core chapters of the book.

Objectives of the Project Structure

In the beginning, Android apps tended to be pretty simple, as we only had a handful of devices, a smattering of users, one primary distribution channel (the then-Android Market) and few major investors in the Android ecosystem.

Times have changed.

Now, Android apps for public consumption can be terribly complex, let alone apps for internal enterprise use (which seem to be complex as a side effect of being developed by an enterprise). We have multiple distribution channels, such as the Amazon AppStore for Android and Yandex.Store. We have a billion devices and nearly a billion users. Brands large and small are flocking to Android, bringing with them their own challenges.

The new build system is designed to simplify creating complex Android applications, while, ideally, not making simple Android applications a lot harder. It is designed for scenarios like:

- Supporting multiple distribution channels, which may require multiple in-app purchasing engines
- Supporting one app that is customized for individual clients, such as for use by different enterprises
- Supporting an app that really needs to have different APKs for different types of devices, despite all efforts to support all devices from a single APK
- Supporting an app that is part of a much larger integrated system and needing to be built as part and parcel of that larger system
- Supporting a fleet of apps that depend upon common code, resources, third-party libraries, and the like
- And so on

The new project structure, coupled with the Android Gradle Plugin and Gradle itself, makes all of this possible... albeit with a bit of a learning curve.

Terminology

To understand what the new project structure entails, we need to define a few terms, from Gradle and the Android Gradle Plugin.

Source Sets

To quote the Gradle documentation: "A source set is simply a group of source files which are compiled and executed together." Here, "source" means all the inputs that you are creating for the app, such as Java source code, Android resources and manifest files, and the like. This is in contrast to *dependencies*, which are inputs that you are (usually) obtaining from other developers, such as reusable libraries.

source sets, on their own, have no particular semantic meaning. You can elect to have your project use a single source set, or several source sets, for organizing your code. You might have different source sets for:

- Production code versus test code, replacing the separate test project that we historically used in Android development
- Interface code versus implementation code
- Different major functional areas within the app, particularly if they are maintained by separate teams or developer pairs
- And so on

As we will see, the new project structure assumes the existence of at least one source set, typically named `main`, but other features of the new build system will involve additional source sets.

Build Types

A build type is one axis of possible alternative outcomes from the build process.

By default, the Android Gradle Plugin assumes that you want two build types: release and debug. These may go through somewhat different build steps, such as applying ProGuard to the release build but skipping it for the debug build.

The Android Gradle Plugin though allows build types to have slightly different configurations, such as adding a `.debug` suffix to the APK's package name, so that you can have a release build and a debug build of your app on the same device at the same time. You also can create new build types for handling different scenarios. The new build system documentation, for example, suggests a separate "jnidebug" build type, where you can indicate that the Linux `.so` files for a project should be compiled in debug mode.

As we will see, creating a new build type involves modifications to the `build.gradle` file and adding a matching source set.

Product Flavors

A build type is one axis for varying your output. A product flavor is another, independent axis for varying your output.

Product flavors are designed for scenarios where you want different release output for different cases. For example, you may want to have one version of your app built

to use Google's in-app purchasing APIs (for distribution through the Play Store) and another version of your app built to use Amazon's in-app purchasing APIs (for distribution through the Amazon AppStore for Android). In this case, both versions of the app will be available in release form, and you may wish to have separate debug builds as well. And *most* of the code for the two versions of the app will be the same. However, you will have different code for the different distribution channels — not only does the right code have to run for the right channel, but there is no particular value in distributing the code for one channel through the other channel.

Another example would be an app that is branded and configured for different enterprise customers. You see this a lot with Web apps — the vendor sells a branded-and-configured version of the Web app to the customer, whether that app runs on vendor-supplied hardware or customer-supplied hardware. Similarly, the maker of an Android app for collecting employee timesheets might want to offer to its customers for *their* version of the timesheet app to sport the customer's logo, tie into the customer's specific accounting server, enable or disable features based upon how the customer uses timesheets, and so on. However, *most* of the code is shared between all customers, and so when the app is updated to add features or fix bugs, new builds are needed for *all* of the customers. In this case, each customer can be set up as an independent product flavor, sharing much of the code, but with slightly different code, resources (e.g., logo), and configuration based upon that customer's needs.

Product flavors are optional. If you do not describe any product flavors in your `build.gradle` file, it is assumed that you have a single product flavor, referred to internally as `default`. Many apps will not need product flavors; this is a feature that you will opt into as needed.

As we will see, creating a new product flavor involves modifications to the `build.gradle` file and (usually) adding a matching source set.

Build Variants

The term "build variant" is used for the cross product of the build types and the product flavors. So, a project with `debug` and `release` build types and `google` and `amazon` product flavors will result in a total of four build variants by default:

1. `debug` + `google`
2. `debug` + `amazon`
3. `release` + `google`
4. `release` + `amazon`

Flavor Dimensions

Sometimes, even this is insufficient flexibility, such as the `google` and `amazon` scenario described earlier in this section. Or, you might need separate `free` versus `paid` editions, if you want to have an up-front fee for accessing a premium version of your app.

Product flavors are considered to be part of a "flavor dimension". All flavors have to be a part of a flavor dimension, starting with Android Studio 3.0 — previously, they could be part of a default flavor dimension. You can have as many flavor dimensions as you need (e.g., one for free versus paid, one for distribution channel).

These then add another factor into the cross-product that determines your build variants. Suppose we have a `dist` flavor dimension, consisting of `free` and `paid` product flavors, and we have a `channel` flavor dimension, consisting of `google` and `amazon` flavors. Now, we have a total of *8* possible build variants, when we factor in the build types:

1. `debug` + `google` + `free`
2. `debug` + `amazon` + `free`
3. `release` + `google` + `free`
4. `release` + `amazon` + `free`
5. `debug` + `google` + `paid`
6. `debug` + `amazon` + `paid`
7. `release` + `google` + `paid`
8. `release` + `amazon` + `paid`

Configuring the Stock Build Types

The `debug` and `release` build types are ready "out of the box" for your use, with a reasonable set of defaults. However, you can change those defaults and make other adjustments to how those build types work, in addition to defining your own build types. Here, we will look at the options for changing the behavior of any build type, focusing on the stock `debug` and `release` build types.

Source Sets

Each build type can have its own associated source set. If you skip the directory for it, that means that the build type is not contributing changes to the `main` source set.

For example, later in the book, we will take a look at Stetho, a diagnostic library that you can add to your Android app to help understand what is going on inside of it. This code is handy during development, but it is *not* the sort of thing that you want to be shipping in production. Instead, you want to use it only for `debug` builds.

So, in the `Diagnostics/Stetho` sample project, we have not only a `main` source set, but also a `debug` source set alongside `main`. In `debug`, we have an `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  package="com.commonsware.android.stetho"
  android:versionCode="1"
  android:versionName="1.0">

  <application
    android:name="com.commonsware.android.stetho.StethoApp"
    tools:replace="android:name">
  </application>

</manifest>
```

(from Diagnostics/Stetho/app/src/debug/AndroidManifest.xml)

The `com.commonsware.android.stetho.StethoApp` Java class referenced in the `<application>` element also resides in the `debug` source set.

We will see how these blend in with the contents of our `main` source set later in this chapter.

build.gradle Settings

We can also use the `buildTypes` closure in `build.gradle` to configure the behavior of the `debug` and/or `release` build types.

For example, a very popular thing to use is `applicationIdSuffix`, to append a value onto the application ID for builds of a particular build type. This allows the application ID for that build type to be different than the application ID of other build types of the same app. In other words, you can have `debug` and `release` builds installed on the same device at the same time.

As was noted earlier in the book, the `defaultConfig` closure allows us to change aspects of what is found in the `AndroidManifest.xml` file, replacing anything found in the actual file from the `main` source set.

Similarly, the `buildTypes` closure is where we configure the behavior of the build types. Each build type to be configured gets its own closure inside of `buildTypes`, and in there we can override various properties.

Notable properties that we can specify for a build type include:

- `debuggable` (to override `android:debuggable` from the `<application>` element in the manifest, to indicate that the app should be considered debuggable)
- `applicationIdSuffix` (to append to the package name specified by the manifest or the `defaultConfig` `applicationId` property)
- `versionNameSuffix` (to append to the version name specified by the manifest or the `defaultConfig` `versionName` property)

So, we can have:

```
buildTypes {
    debug {
        applicationIdSuffix ".d"
    }
}
```

Our `debug` build type adds suffixes to the application ID role for the package name. Note that altering the application ID *only* affects the package name as seen by Android when the app is installed and when the app is run. It does *not* affect the directory in which the `R` class is built, which uses the package name from the `AndroidManifest.xml` file. It also does not affect any of the Java packages for our own classes, which are whatever we used when we wrote them. Hence, much of our code will be oblivious to the package name change. However, if you want to reference the *real* package name, such as for looking things up in `PackageManager` or for use with constructing a `ComponentName`, use `getPackageName()` on any `Context` (like an `Activity`), rather than some hard-coded string, as `getPackageName()` returns what the runtime environment thinks the package is, which will include any suffixes added during the build process. Or, use `BuildConfig.APPLICATION_ID`, in cases where you do not have a `Context` handy on which to call `getPackageName()`.

We can also have a `signingConfig` property, for configuring how our APK files are digitally signed. This will be covered in a later chapter.

Order of Precedence

Properties defined for a build type, and the properties defined for the `defaultConfig` will override their equivalents in the `AndroidManifest.xml` file. However, a build type's source set can *also* have its own `AndroidManifest.xml` file. The overall order of precedence is:

- What is in `build.gradle` takes precedence over...
- ...what is in a build type's `AndroidManifest.xml` file, which takes precedence over...
- ...what is in the `main` `AndroidManifest.xml` file

However, merging manifests in general is a complex topic, with a separate chapter later in this book.

Resources from the build type's source set are merged into the resources from the `main` source set, and if there are collisions, the build type's resource takes precedence. The same is true for assets.

However, the behavior of Java source is slightly different. The build type's source set is still merged with the `main` source set, but if there is a collision, the result is a build error. Either the build type or the `main` source set can define any given source file, not both. So, while `debug` could have one version of `your/package/name/Foo.java` and `release` could have a different version of `your/package/name/Foo.java`, `main` could not *also* have `your/package/name/Foo.java`. Hence, if you define a class in a build type, you may need to define that class in *all* build types, so that any references from `main` to that class are satisfied for all build types.

One case where this would not be required would be for debug-only activities. Suppose that you wanted an activity in your app to provide diagnostic information to developers of that app regarding the state of caches and other in-memory constructs. While you could get at that stuff via a debugger, that is sometimes annoying, and just tapping on a launcher icon can be easier. But you do not want, let alone need, this diagnostic activity in your release builds. To make this work, you would put the activity's Java class only in the `debug` source set, along with its resources and manifest entry (complete with `MAIN/LAUNCHER` `<intent-filter>`). Since the `main` source set does not refer to your diagnostic activity, there is no requirement for the `release` build type to have an implementation of that Java class.

Adding Build Types

Many developers will fare just fine with the `debug` and `release` build types, perhaps with some adjustments as shown above. A few developers, though, will have other scenarios that warrant new build types. Fortunately, adding a new build type is rather easy.

First, pick whether you want to have the new build type start with the settings from `debug` or `release`. Then, use `initWith` to create a new build type initialized from that existing build type:

```
    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
        }

        dogfood.initWith(buildTypes.debug)

        dogfood {
            applicationIdSuffix ".dawg"
        }
    }
```

Here, we define a new `dogfood` build type, with its own application ID suffix.

As with the built-in build types, your new build types can have their own source sets, by adding the appropriately-named directories underneath `src/`.

Now, we gain Gradle tasks with `Dogfood` in the name, like `installDogfood`, to go along with their `Debug` and `Release` counterparts.

Adding Product Flavors and Getting Build Variants

Many apps will not need product flavors, but some will. Adding a product flavor is similar, in many respects, to adding a build type.

This involves two steps:

1. Having a `flavorDimensions` statement in the `android` closure, to define the flavor dimension(s) that you wish to use

2. Having a `productFlavors` closure where you define the flavors and their properties, including what dimension they belong to

```
flavorDimensions "channel"

productFlavors {
    goog {
        dimension "channel"
        applicationId "com.commonsware.android.awesomeapp.goog"
    }

    amzn {
        dimension "channel"
        applicationId "com.commonsware.android.awesomeapp.amzn"
    }
}
```

Here, we define one flavor dimension (`channel`) and two product flavors (`goog` and `amzn`), assigning a distinct `applicationId` to each of those flavors.

The `defaultConfig` is implemented using the same object type as is used for product flavors. Hence, we can configure the same things on a product flavor that we can on the `defaultConfig`, such as `applicationId`, as is done in this `build.gradle` file.

We could have source sets for each of these as well. For example, the `goog` source set might contain code that is used only for Google Play ecosystem devices (e.g., Play Services), while `amzn` might contain code used for devices distributed by other channels.

In terms of order of precedence:

- Product flavors override the `main` source set and the `defaultConfig`
- Build types override the product flavors

Our task names get more numerous and more complicated, to reflect the cross product of the product flavors and build types. Now, rather than `installDebug`, and `installRelease`, we have:

- `installAmznDebug`
- `installAmznRelease`
- `installGoogDebug`
- `installGoogRelease`

Doing the Splits

The Android Gradle Plugin offers `splits` as a lightweight canned replacement for product flavors for two scenarios:

- Having different APK files with different NDK binaries for ARM vs. x86 (vs. anything else)
- Having different APK files with resources for a specific screen density, important for those apps that have so many graphics that they are bumping up against distribution channel limits (e.g., 100MB on the Play Store, up from an earlier 50MB limit)

All you as a developer do is request that a particular split be enabled, with limited configuration. Notably, you do not have separate Gradle configuration (e.g., `applicationId`) nor source sets for splits. That allows splits to be processed more quickly at build time, as the build tools can make some simplifying assumptions and avoid a lot of recompiling.

Scoping Your Splits

A split, by default, will generate one APK per *possible* type of output. For example, splitting on density will give you one APK for `ldpi`, `mdpi`, `tvdpi`, `hdpi`, `xhdpi`, `xxhdpi`, and `xxxhdpi`. Plus, in the case of density, you also get one "universal" APK containing support for all densities by default.

That's nice... but what if you do not need separate APKs for all of those densities? For example, if you do not ship `tvdpi` resources, there is little reason to set up an APK for it separate from, say, the `hdpi` APK.

There are two basic patterns to controlling the scope of what gets built:

1. Use an `exclude` statement to start with the defaults and remove some options
2. Use a `reset()` method to wipe out the defaults, then use an `include` statement to list what you want

In other words, `exclude` implements a blacklist, and the `reset()/include` combination implements a whitelist. All else being equal, a whitelist is probably a better choice, so you can explicitly line it up with what you have written in your app.

Requesting NDK Splits

In your `android` closure, you can add a `splits` closure, containing an `abi` closure, which in turn sets up the APK splits by CPU architecture:

```
splits {
  abi {
    enable true
    reset()
    include 'armeabi-v7a', 'x86'
    universalApk true
  }
}
```

Here, we:

- Enable the split (`enable true`)
- Remove the default ABIs to be included (`reset()`)
- List the ABIs that we want to be included (`include 'armeabi-v7a', 'x86'`)
- Request that a "universal APK" also be created, containing all ABIs (`universalApk true`)

The latter would be useful for distribution channels that do not allow you to upload multiple APK files for different CPU architectures. This way, you can at least distribute your app there, even if it takes up more disk space than you like. By default, for the CPU architectures, you do not get a "universal APK".

Requesting Density Splits

The same basic pattern can be implemented for densities:

```
splits {
  density {
    enable true
    reset()
    include 'hdpi', 'xhdpi', 'xxhdpi'
  }
}
```

Once again, we enable the split, reset the defaults, then opt into the densities that we want.

Note, though, that a "universal APK" is always generated for densities. We do not need to have `universalApk true`, and it would appear that `universalApk false` is not an option at the present time.

Gradle and Android Studio

While most of the work involving build variants revolves around the module's `build.gradle` file, there are some things to consider when working with build variants within Android Studio.

The Build Variants View

When we run our project, Android Studio does not prompt us for a build type or a product flavor. It just runs the project. This begs the question of how Android Studio is determining which build variant is the one to run.

This is handled by the Build Variants view, usually docked on the left side of the Android Studio IDE window:

![Build Variants View, for a Simple Project](andproj-source sets3.png)

Each of your app's modules is shown, along with the current build variant that will be used if you run that module. Tapping on the build variant will allow you to choose an alternative build variant:

![Build Variants View, for a Project with Custom Build Types and Product Flavors](andproj-source sets4.png)

The Android Project View

Earlier in the book, when introducing Android Studio, we saw the Android project view. Elsewhere, we saw how the Android project view can help you manage resources across multiple resource sets.

Just as the Android project view "collapses" resource set, it also collapses source sets:

![Android Project View, Showing Java Source](andproj-source sets1.png)

Here, we have two editions of the `com.commonsware.myapplication` package. One is just the package name, while the other has "(androidTest)" appended to it. That, as you might imagine, reflects the `main` and `androidTest` source set, respectively:

![Classic Project View, Showing Java Source](andproj-source sets2.png)

This may be a bit useful between `main` and `androidTest`. It is likely to be far more useful if you employ product flavors, as your classes for the flavors will appear side-by-side... at least for the currently-selected flavor in the Build Variants view.

Flavors, Build Types, and the Project Structure Dialog

You are welcome to use the Build Types and Product Flavors tabs in the project structure dialog to maintain these portions of your `build.gradle` file, at least for simpler scenarios.

Manifest Merger Rules

When Android library projects were added as an option for app development, one problem became apparent: while libraries could contribute code and resources, they could not contribute manifest entries. Developers using libraries would sometimes have to add elements to their app manifest at the request of library authors, to add permissions, define components, and the like.

The Android Gradle Plugin has a robust set of rules for "manifest merger". While the term "manifest merger" is still used, in reality, the Android Gradle Plugin synthesizes a manifest for your app from a variety of sources, including apps, libraries, and `build.gradle` files, also varying based upon build types and product flavors.

This chapter will help to explain a bit more about what is possible what the rules are for the manifest merger process.

Prerequisites

Understanding this chapter requires that you have read the chapters that introduce Gradle and cover basic Gradle/Android integration, including build variants.

Manifest Scenarios

You might be wondering "why do we need all of this?" That is a fair question. Certainly, we were able to get by for quite a while without this sort of flexibility and accompanying confusion.

Here are some scenarios which help explain what you will get out of the manifest merger capabilities.

Library Manifest and App Manifest

A library — whether one of yours or one obtained via an AAR artifact from some repository — may need to augment the app's manifest. For example:

- a library for an ad network might require the `INTERNET` permission, so that apps that do not directly use the Internet still wind up requesting that permission
- a library providing a canned "about" activity might want to inject the `<activity>` element that you can use without requiring the developer to add it manually
- a library needs to be able to specify the `minSdkVersion` that it requires, which might supersede the value specified by the app, so the combined whole uses the most conservative value

App Manifest and Build Types

You may have particular needs for your main application that vary based upon build type that affect the manifest versus other things (e.g., ProGuard configuration).

For example, it may be that in debug builds, you want to have an activity that you can bring up, perhaps through `adb shell am`, that will give you diagnostic information about the app itself, or starts some diagnostic service that you can then access through your development machine's Web browser. In this case, that activity and that service would only be desired in `debug` builds, not `release` builds. And while the activity and the service code would simply be in the `debug` source set, you also need to merge in the manifest `<activity>` and `<service>` elements, plus perhaps other things (e.g., extra `<uses-permission>` elements that those diagnostic components need but the rest of the app does not).

App Manifest and Product Flavors

Product flavors can override values from the `defaultConfig`, such as defining distinct `applicationId` values, and that needs to be taken into account in the combined app.

Also, product flavors might need their own manifest entries to accommodate distribution channel-specific APIs, such as swapping between Play Services and Amazon equivalents for in-app purchasing or maps.

Combo Platters

And, of course, you may have some mix of all of the above.

Pieces of Manifest Generation

When you build your app, the build tools will combine information from all of the aforementioned sources to synthesize "one true manifest" that is used for the build.

However, there may be overlaps in what the sources provide, such as both a library and the app specifying a `minSdkVersion`. Hence, there are some basic rules and control structures that you have to manage the generation process, at least somewhat.

Merger Rules

Generally speaking:

- the manifests and Gradle configurations for product flavors and build types will override...
- the app's `main/` manifest, which will override...
- the manifest of any libraries

Libraries will be considered in the order of declaration — in other words, the order that they appear in the `dependencies` closure. This includes transitive dependencies, where one dependency requires another dependency, though the exact rules here are presently unclear.

For any given element or attribute, there are specific rules for how conflicts are resolved. We will explore those later in this chapter.

Note that values in `build.gradle`, such as the `defaultConfig` closure and its `minSdkVersion` and such, trump everything that result from the merger of disparate manifests from different sources.

Markers and Selectors

Through `tools:` attributes in the manifests that you control (i.e., not manifests from third-party libraries), you will be able to override the default rules for conflict resolution.

For example, a library might declare a particular theme to use for an activity that it publishes. That might be a reasonable default theme, but you may wish to override that theme in your app. A `tools:replace` attribute, in your `<activity>` element, will be able to teach the build tools that your `android:theme` value should replace the one from the library, whereas normally a conflict on an attribute like this would result in a build error.

You can also use "selectors" to help control in which scenarios a particular marker is applied, such as applying a marker only for a conflict arising from a specific library.

These markers and selectors will be explored in greater detail later in this chapter.

Placeholders

Sometimes, the "merger" we want to do involves something more involved.

For example, the `applicationId` and `applicationIdSuffix` properties that we set in various places in `build.gradle` can be used to allow for different variants of our builds to be installed at the same time on the same device. However, that is only true some of the time. If an app publishes a `ContentProvider`, not only does the application ID have to be unique, but so does the authority (or authorities) supported by that `ContentProvider`. This is not handled automatically, and so even though you might have the application ID distinct for different build variants, they still would conflict at install time because their provider authorities were the same.

The manifest generation process supports the notion of placeholders, where for string values in the manifest — like the `android:authorities` attribute on a `<provider>` — you can "splice in" dynamic values. One dynamic value that you can use "out of the box" is your build variant's `applicationId`, so you can have something like:

```
android:authorities="${applicationId}.provider"
```

to have the authority for the provider match the build variant's `application`, but with a `.provider` suffix.

We will explore the rules around these placeholders later in this chapter.

Examining the Merger Results

The generated manifest, combining the contents of all the manifests from the app, build types, product flavors, and libraries, will wind up in the `build/intermediates/manifests/` directory of your module (e.g., `app/`). Inside that directory will be subdirectories associated with each build variant, and in those subdirectories reside the generated manifest for that build variant.

The `Manifest/Merger` sample application is designed to illustrate how these merger rules work. Note that the application does not run — it exists merely to show the results of building the APK and, along the way, generating the manifests.

This project contains an app module (`app/`) and a library module (`lib/`), with the app depending upon the library. The app module has source sets for both `main/` and `debug/`, the latter for debug builds. The app module also defines two product flavors, `chocolate` and `vanilla`, with a source set for `vanilla/`. All three source sets (`main/`, `debug/`, `vanilla/`) have their own `AndroidManifest.xml` files. Adding in the manifest from the library, and you have four manifests in total that may be used to create the manifest for the app. In particular, for a `vanilla debug` build, all four manifests will be relevant and merged together.

If you build the project, particularly via the `gradle` command, you will get manifests based on what builds you create. For example, `gradle assembleVanillaDebug` will create a generated manifest in `build/intermediates/manifests/vanilla/debug/`.

As you are trying to determine how manifest merging is working in your project, you may find it useful to peek at these generated manifests from time to time... as we will here in this chapter.

Viewing Merged Manifests in Android Studio

Far and away the easiest way to see the effects of manifest merger on your app is to use Android Studio 2.2 (or higher) and its manifest merger viewer.

When you open a manifest in Android Studio, there are two sub-tabs. One, named "Text", has the XML editor where you define the manifest contents. The other, named "Merged Manifest", shows the results of the manifest merger process, using whatever build variant you have chosen for the module in the Build Variants tool.

The “Merged Manifest” tab has two panes: a tree of XML on the left, and a legend on the right:

Figure 332: Merged Manifest Tab

The contributors to the merged manifest are color-coded, to help make it easy for you to see where a particular element or attribute came from. You can get additional details by clicking on attributes or elements, as we will see when we work through some examples throughout this chapter.

Merging Elements and Attributes

Different sources of manifest data can contribute elements to the generated combined manifest. In many cases, these elements do not conflict, such as a library contributing a `<uses-permission>` element to an app. However, sometimes, what one source of manifest data wants is different than what another source of manifest data wants, and for that, we need to settle out what the generated manifest will contain.

Basic Merger Rules

Many element names can appear several times in a manifest, such as multiple `<uses-permission>` elements. Many of those have an identifier, usually `android:name`, that distinguishes one from the next. In general, if two manifest sources both contribute the same element (i.e., same element name, same `android:name` value), those two elements are themselves merged, which means:

- Any attributes that are in one, but not the other, are added to the combined element
- Any attributes that are in both, and are not identical in value, result in a merge conflict compile error, unless the resolution is specified via a marker
- All child elements (e.g., `<intent-filter>` inside of an `<activity>`) are merged, applying the same rules

Of course, if one manifest supplies a specific element instance, and others do not, then the specific element instance is simply included without worrying about any other merge logic.

Singleton elements — ones that could only ever appear once in the manifest — are treated as matching if they exist in more than one manifest. So, for example, the `android:versionCode` and `android:versionName` attributes of the `<manifest>` element are merged, as are attributes of `<support-screens>`, each of which can only exist once.

Example #1: Manifest Attributes

The `main/` version of the manifest defines an `android:versionName` attribute:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.merger"
  android:versionName="1.0 Main">

  // other stuff here

</manifest>
```

None of the other manifest versions do. Hence, the `main/` version of the manifest "wins", and its `android:versionName` is used (only to perhaps be overridden by `build.gradle` values):

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.merger.vanilla"
  android:versionCode="1"
  android:versionName="1.0 Main" >

  // other stuff here

</manifest>
```

Here, we are showing the results of building the `vanilla` `debug` version of the app, so the package name reflects the `applicationId` defined in `build.gradle` for the `vanilla` product flavor:

```
productFlavors {
  vanilla {
    applicationId "com.commonsware.android.merger.vanilla"
  }

  chocolate {
    applicationId "com.commonsware.android.merger.chocolate"
  }
}
```

Similarly, the `versionCode` shows up because it is defined in `build.gradle`:

```
  defaultConfig {
    applicationId "com.commonsware.android.merger"
    minSdkVersion 15
    targetSdkVersion 19
    versionCode 1
  }
```

However, since `build.gradle` did not specify `versionName`, the version name comes from the manifests.

If another manifest also defined `android:versionName`, its value would need to match that of the one in `main/`, or you will get a build error from the Android Gradle Plugin... unless you use a marker, described later in this chapter.

The "Merged Manifest" view shows `android:versionName` as coming from the `main` manifest. However, as of Android Studio 2.2, contributions from Gradle are not color-coded separately, so we cannot distinguish that the `android:versionCode` really came from Gradle, not from a manifest... unless you click on the attribute:

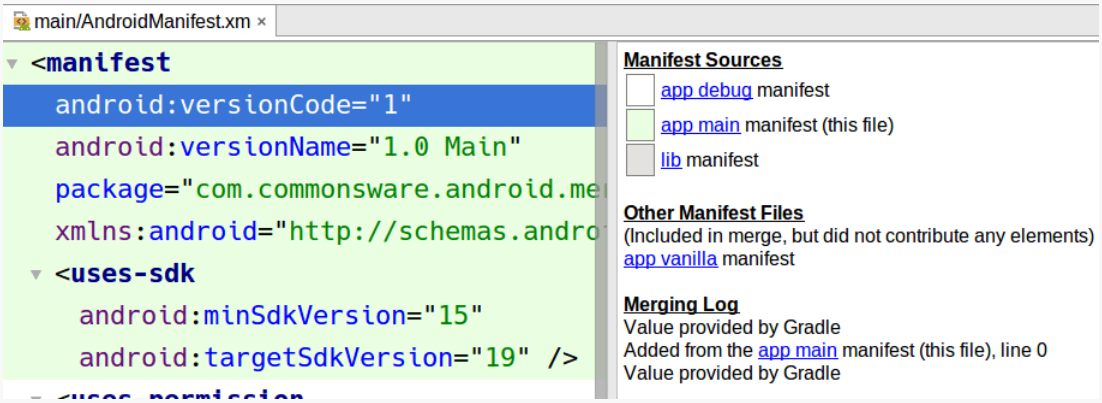


Figure 333: Merged Manifest View, Showing Merged Values from Gradle

Example #2: Additional Permissions

The `debug/` version of the manifest has a `<uses-permission>` element:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

So does the `vanilla/` version, though its has `android:maxSdkVersion` set to 18:

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  android:maxSdkVersion="18"/>
```

The manifest you get from a `vanilla` debug build has the `android:maxSdkVersion` attribute:

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  android:maxSdkVersion="18" />
```

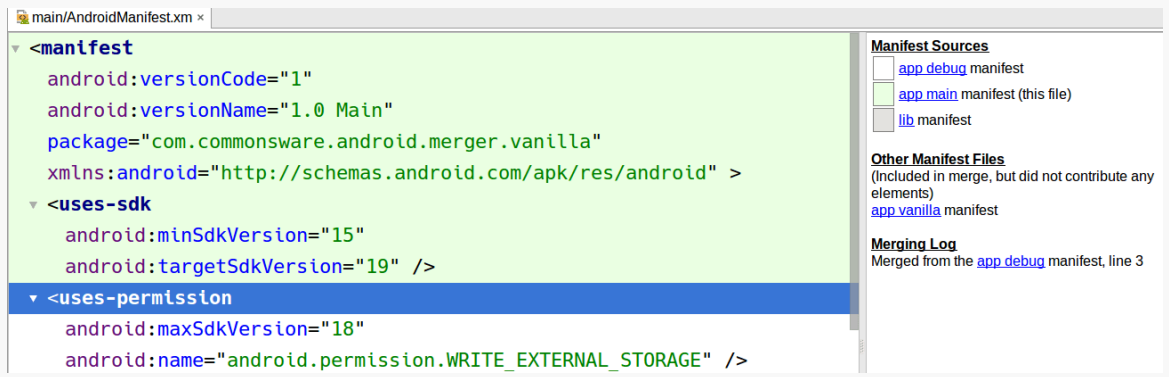


Figure 334: Merged Manifest View, Showing Merged Element

Example #3: Additional Components

The `lib/` version of the manifest has an `<activity>`:

```
<activity android:name="ThisActivityDoesNotExist">
  <intent-filter>
    <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

This is contributed by the library (or would be, if there actually was source code for the activity...). Neither the `main/` nor the `debug/` source set defines it, and so it is included verbatim in the result for `chocolate` builds:

```
<activity android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist" >
  <intent-filter>
    <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />

    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Note that since the `android:name` attribute had a bare class name, the generated manifest expands that to include the *library's* package name (`com.commonsware.android.merger.lib`). Note that this is the package name defined in `AndroidManifest.xml` — you cannot have an `applicationId` in `build.gradle` for a library project.

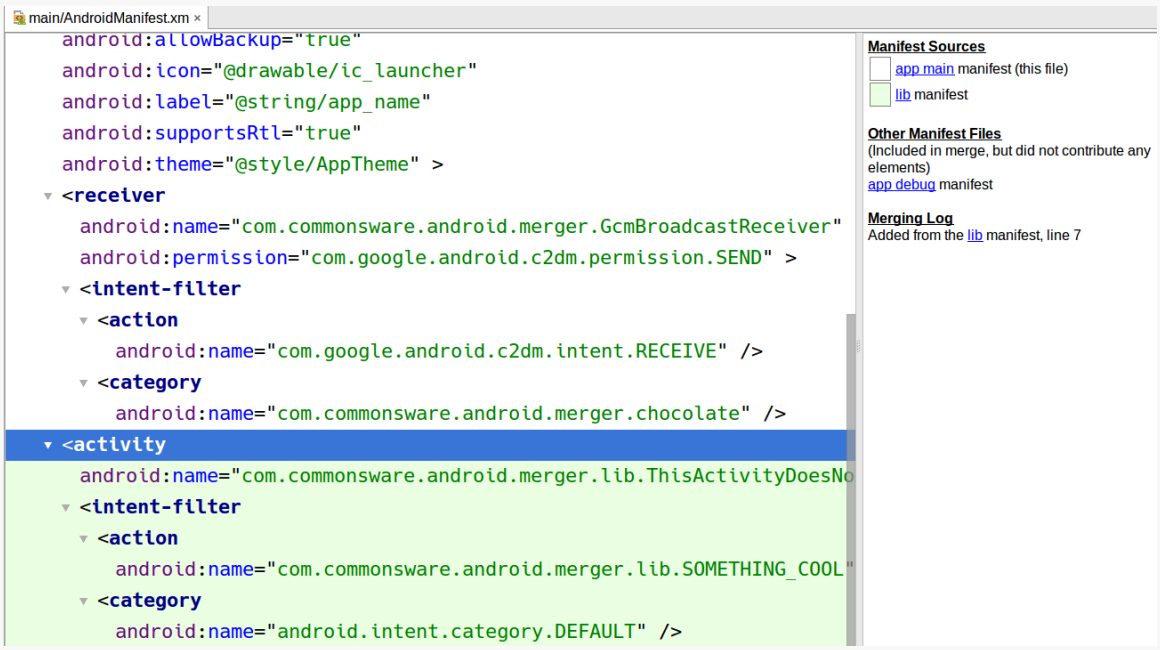


Figure 335: Merged Manifest View, Showing Merged Element

Example #4: Intent Filter

However, the `vanilla/` manifest also defines the same activity, this time with another `<intent-filter>`:

```
<activity android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist">
  <intent-filter>
    <action android:name="com.commonsware.android.merger.SOMETHING_VANILLA" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Note that here, we need to have the fully-qualified class name, as we are trying to affect the library-supplied activity.

In a `vanilla` build, *both* `<intent-filter>` elements will be included by default:

```
<activity android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist" >
  <intent-filter>
    <action android:name="com.commonsware.android.merger.SOMETHING_VANILLA" />

    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
```

```
  <intent-filter>
    <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />

    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

This allows an app developer to add new ways of accessing an activity (or other component) exposed by a library.

However, the merged manifest view may not necessarily indicate where the elements came from properly:

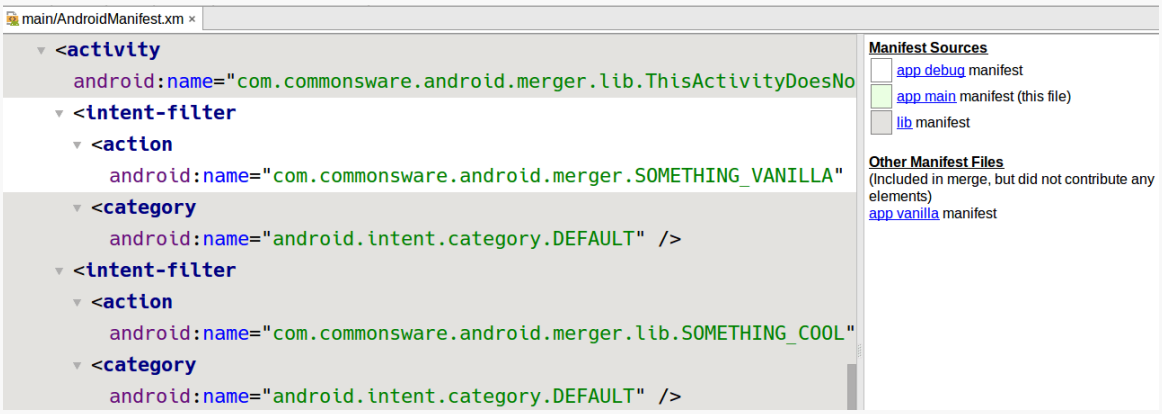


Figure 336: Merged Manifest View, Showing Merged Element

Some Unusual Scenarios

Not everything fits the neat-and-tidy rules from the above sections and require special explanation.

uses-sdk

The `android:minSdkVersion` and `android:targetSdkVersion` from the highest-priority manifest will be used. If, however, a library's manifest specifies higher values for `minSdkVersion`, you will get a build error.

Hence, it is incumbent upon library authors to correctly assess how old a version of Android they are able to support, setting `android:minSdkVersion` as low as possible.

Conversely, library authors should aim to support either old or new behavior that is controlled by `android:targetSdkVersion`. For example, a library that uses `AsyncTask` should not assume that the `android:targetSdkVersion` is below 13 and therefore `execute()` will result in multi-threaded behavior on Android 3.2+. Instead, the library should use `executeOnExecutor()` on API Level 11+ devices, to specifically opt into the multi-thread thread pool, as this avoids any behavior changes based upon `android:targetSdkVersion`.

uses-feature and uses-library

The `android:required` attribute is logically OR'd among all contributors of a `<uses-feature>` element for a specific `android:name` value. In other words, if any contributor says that the feature is required, it is required. Otherwise, if one or more contributors ask for the `<uses-feature>` element but say that it is not required, it is put in the combined manifest with `android:required="false"`.

The under-utilized `<uses-library>` uses the same rule for handling the merger of its `android:required` attribute.

Markers and Selectors

Sometimes, the default merger rules will not work to your satisfaction. In particular, when there are conflicts, the build will fail, and probably that is not a desired outcome.

To declare who wins in the case of conflicts, you can use `tools:*` attributes in the manifest elements. Specifically:

- `tools:node` indicates how to resolve a conflict between two editions of this particular XML element (e.g., an `<activity>` for the same `android:name`)
- `tools:replace` indicates that certain attributes from a lower-priority edition of the manifest should be overwritten by their replacement values from a higher-priority edition of the manifest
- `tools:remove` indicates that certain attributes from a lower-priority edition of the manifest should be removed entirely

Each of these, being in the `tools` namespace, will require you to have `xmlns:tools="http://schemas.android.com/tools"` on the root `<manifest>` element, if it is not there already. These attributes only affect the build tools and

have no runtime implications, other than in terms of how the build tools build your app based on the `tools` attributes.

For example, the main manifest has `android:supportsRtl="true"` on the `<application>` element:

```
<application android:allowBackup="true"
  android:label="@string/app_name"
  android:icon="@drawable/ic_launcher"
  android:theme="@style/AppTheme"
  android:supportsRtl="true">

  // other stuff here

</application>
```

For a project with a `targetSdkVersion` of 17 or higher, `android:supportsRtl="true"` enables automatic mirroring support for your layouts for right-to-left (RTL) languages.

The vanilla manifest wants to override this, replacing the value with `false`, as perhaps the code in that flavor is not yet ready for automatic mirroring. However, if the vanilla manifest just had `android:supportsRtl="false"` in its `<application>` element, the build would fail, as that value conflicts with the one in the main manifest. Hence, the vanilla manifest also needs to indicate that its `android:supportsRtl` value should replace the original one, via a `tools:replace` attribute:

```
<application android:allowBackup="true"
  android:label="@string/app_name"
  android:icon="@drawable/ic_launcher"
  android:theme="@style/AppTheme"
  android:supportsRtl="false"
  tools:replace="android:supportsRtl">

  // other stuff here

</application>
```

In the output, `android:supportsRtl="false"` wins:

```
<application
  android:allowBackup="true"
  android:icon="@drawable/ic_launcher"
```

```
  android:label="@string/app_name"
  android:theme="@style/AppTheme"
  android:supportsRtl="false" >

  // other stuff here

</application>
```

Both `tools:replace` and `tools:remove` take a comma-delimited list of attributes that should be affected by that rule.

Once again, the merged manifest view may not indicate the source of the attribute properly:

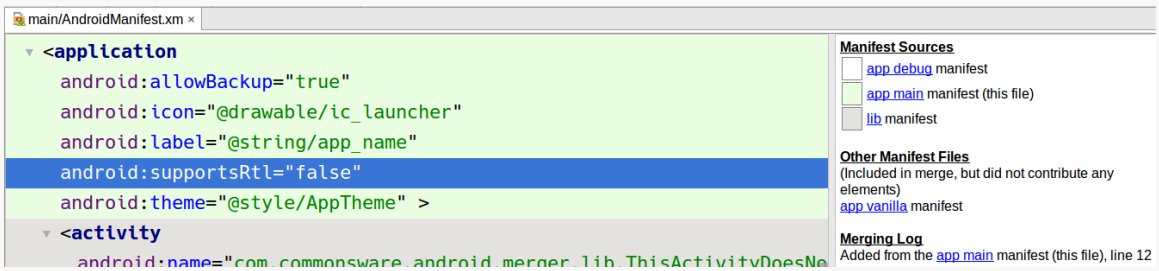


Figure 337: Merged Manifest View, Showing Merged Attribute

The `tools:node` attribute affects the entire XML element in which it resides. There are five primary values for `tools:node`:

1. `merge`, which is the default behavior described by the merger rules earlier in this chapter
2. `replace` says that the lower-priority manifest's version of this element should be replaced in its entirety with the higher-priority manifest's version of this element
3. `merge-only-attributes` says that the lower-priority manifest's version of this element should have its attributes replaced by the ones from the higher-priority manifest's version of this element, but child elements (e.g., an `<intent-filter>` underneath the annotated `<activity>` element) are left alone
4. `remove` says that the lower-priority manifest's version of this element should be removed without any replacement
5. `removeAll` says that all elements of this name (e.g., `<uses-permission>`) from lower-priority manifests should be removed, regardless of scopes like `android:name`

There is also a `strict` value that indicates that any duplication, even if it could be successfully merged, should result in a build failure. Most likely, this would be used sparingly.

By default, these `tools` attributes affect all manifests. However, it could be that you only want to affect a specific manifest, such as one coming from a certain library. In that case, `tools:selector`, in the same XML element as the other `tools:*` attributes, provides the package name of the library that the other `tools:*` attributes affect.

Employing Placeholders

The Google Cloud Messaging (GCM) system has some unusual requirements for the manifest of apps that use GCM:

- The app needs to define a custom permission, based on the application ID, via a `<permission>`
- The app needs to hold that custom permission, via a `<uses-permission>` element
- The app needs to have a `BroadcastReceiver` whose `<intent-filter>` has a `<category>` whose name is the application ID

Hence, in a GCM client app's manifest, there are three places where the application ID needs to appear. This needs to be the app's actual application ID, as may be defined either via manifests or via `applicationId` or `applicationIdSuffix` statements in a `build.gradle` file. Since the application ID can be overridden by those Gradle statements, we cannot just hard-code the application ID into the spots in the manifest.

Fortunately, part of what we get with manifest generation are placeholders.

Placeholders allow us to inject values from `build.gradle` into the manifest, particularly in XML attribute values. An `applicationId` placeholder is available automatically, and we can define custom ones via a `manifestPlaceholders` map.

For example, the main manifest for the sample project uses the `applicationId` placeholder in the requisite locations:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.merger"
    android:versionName="1.0 Main">
```

```
    <permission android:name="${applicationId}.C2D_MESSAGE"
        android:protectionLevel="signature" />
    <uses-permission android:name="${applicationId}.C2D_MESSAGE" />

    <application android:allowBackup="true"
        android:label="@string/app_name"
        android:icon="@drawable/ic_launcher"
        android:theme="@style/AppTheme"
        android:supportsRtl="true">
      <receiver
        android:name=".GcmBroadcastReceiver"
        android:permission="com.google.android.c2dm.permission.SEND" >
        <intent-filter>
          <action android:name="com.google.android.c2dm.intent.RECEIVE" />
          <category android:name="${applicationId}" />
        </intent-filter>
      </receiver>
    </application>

</manifest>
```

The vanilla debug version of the generated manifest replaces those `${applicationId}` placeholders with the actual `applicationId`, such as the following for a `vanilla` build:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.merger.vanilla"
  android:versionCode="1"
  android:versionName="1.0 Main" >

  <uses-sdk
    android:minSdkVersion="15"
    android:targetSdkVersion="19" />

  <uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="18" />

  <permission
    android:name="com.commonsware.android.merger.vanilla.C2D_MESSAGE"
    android:protectionLevel="signature" />

  <uses-permission android:name="com.commonsware.android.merger.vanilla.C2D_MESSAGE"
/>
```

```
  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme"
    android:supportsRtl="false" >
    <activity
android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist" >
      <intent-filter>
        <action android:name="com.commonsware.android.merger.SOMETHING_VANILLA" />

        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />

        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
    </activity>

    <receiver
      android:name="com.commonsware.android.merger.GcmBroadcastReceiver"
      android:permission="com.google.android.c2dm.permission.SEND" >
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />

        <category android:name="com.commonsware.android.merger.vanilla" />
      </intent-filter>
    </receiver>
  </application>

</manifest>
```

Note that the entire XML attribute value does not have to be a placeholder. For example, the `android:name` values for the `<permission>` and `<uses-permission>` elements blend the `applicationId` in with a fixed string: `android:name="${applicationId}.C2D_MESSAGE"`.

If you want additional placeholders, you can define a `manifestPlaceholders` map in `defaultConfig` or in a product flavor:

```
android {
  defaultConfig {
    manifestPlaceholders = [ foo: "bar"]
  }
```

```
  productFlavors {
    vanilla {
    }

    chocolate {
      manifestPlaceholders = [ foo: "baz" ]
    }
  }
}
```

Then, you can refer to any of your custom placeholders via the same `${}` syntax (e.g., `${foo}`, with the proper value being applied during manifest generation.

Signing Your App

Perhaps the most important step in preparing your application for production distribution is signing it with a production signing key. While mistakes here may not be immediately apparent, they can have significant long-term impacts, particularly when it comes time for you to distribute an update.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Role of Code Signing

There are many reasons why Android wants you to sign your application with a production key. Here are perhaps the top three:

- It will help distinguish your production applications from debug versions of the same applications
- Multiple applications signed with the same key can access each other's private files, if they are set up to use a shared user ID in their manifests
- You can only update an application if it has a signature from the same digital certificate

The latter one is the most important for you, if you plan on offering updates of your application. If you sign version 1.0 of your application with one key, and you sign version 2.0 of your application with another key, version 2.0 will not install over the top of version 1.0 — it will fail with a certificate-match error.

What Happens In Debug Mode

Of course, you may be wondering how you got this far in life without worrying about keys and certificates and signatures (unless you are using Google Maps, in which case you experienced a bit of this when you got your API key).

The Android build process creates a debug key for you automatically. That key is automatically applied when you create a debug version of your application (e.g., running the app in your IDE). This all happens behind the scenes, so it is very possible for you to go through weeks and months of development and not encounter this problem.

In fact, the most likely place where you might encounter this problem is in a distributed development environment, such as an open source project. There, you might have encountered the third bullet above, where a debug application compiled by one team member cannot install over the debug application from another team member, since they do not share a common debug key. You may have run into similar problems just on your own if you use multiple development machines (e.g., a desktop in the home office and a notebook for when you are on the road delivering Android developer training).

Finding Your Debug Keystore

The debug keystore is a `debug.keystore` file in your Android SDK data directory. This directory is not where your SDK is installed, but rather is where the tools store data unique to your account on your developer machine, such as your emulator AVDs.

This directory can be found at:

- `~/.android/` on macOS and Linux
- `C:\Documents and Settings\...\.android\` on Windows XP
- `C:\Users\...\.android\` on Windows environments newer than XP

(where `...` is your Windows username)

Synchronizing Your Debug Signing Key

If you have a development team that, for better coordination, should all use the same `debug.keystore`, just pick one and copy it to all team members' development

machines, replacing their generated ones. The `debug.keystore` file is a binary file and should be transferable between operating systems (e.g., from Linux to Windows).

Production Signing Keys

Beyond the debug keystore, though, you will need one for production use. Distribution channels like the Play Store do not accept apps signed with the debug signing key. So, you will need to create a key that *is* acceptable to those channels, plus arrange to use that key when creating your production apps.

How long your production signing key is valid for is important. Once your key expires, you can no longer use it for signing new applications, which means once the key expires, you cannot update existing Android applications. Also, the Play Store requires your key to be valid beyond October 22, 2033. When you create your key, you will indicate how long it should be valid for.

Note that both the debug signing key and its production counterpart are self-signed certificates — you do not have to purchase a certificate from Verisign or anyone. These keys are for creating immutable identity, but are not for creating confirmed identity. In other words, these certificates do not prove you are such-and-so person, but can prove that the same key signed two different APKs.

Creating a Production Signing Key

The mechanics of creating a production signing key depend on whether you will use an IDE (and, if so, which one) or will create one outside of any IDE.

Android Studio

Android Studio has support to create a production signing key as part of its overall process for creating a production-signed APK, which is covered later in this chapter.

Manually

To manually create a production signing key, you will need to use `keytool`. This comes with the Java SDK, and so it should be available to you already.

The `keytool` utility manages the contents of a “keystore”, which can contain one or more keys. Each “keystore” has a password for the store itself, and keys can also have

their own individual passwords. You will need to supply these passwords later on when signing an application with the key.

Here is an example of running keytool:

```
keytool -genkey -v -keystore cw-release.keystore -alias cw-release -keyalg RSA
-validity 10000 -keysize 2048
```

The parameters used here are:

1. `-genkey`, to indicate we want to create a new key
2. `-v`, to be verbose about the key creation process
3. `-keystore`, to indicate what keystore we are manipulating (`cw-release.keystore`), which will be created if it does not already exist
4. `-alias`, to indicate what human-readable name we want to give the key (`cw-release`)
5. `-keyalg`, to indicate what public-key encryption algorithm to be using for this key (`RSA`)
6. `-validity`, to indicate how long this key should be valid, where 10,000 days or more is recommended
7. `-keysize`, for indicating the length of the signing key (2,048 bits recommended, or go higher if you prefer)

If you run the above command, you will be prompted for a number of pieces of information. If you have ever created an SSL certificate, the prompts will be familiar:

```
$ keytool -genkey -v -keystore cw-release.keystore -alias cw-release -keyalg RSA
-validity 10000 -keysize 2048
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Mark Murphy
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:  CommonsWare, LLC
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:  PA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US correct?
  [no]:  yes
```

```
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a
validity of 10,000 days
  for: CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
Enter key password for <cw-release>
  (RETURN if same as keystore password):
[Storing cw-release.keystore]
```

Signing with the Production Key

How you will apply this production signing key to sign your production app again varies by your tool chain.

Android Studio

Start by opening up your project and going to Build > Generate Signed APK from the main menu. This brings up the first page of a signing wizard:

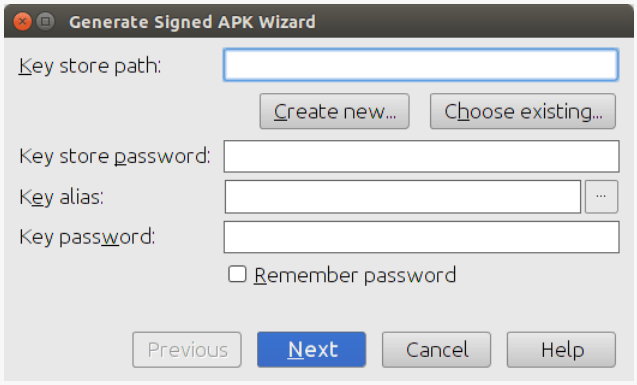


Figure 338: Android Studio Generate Signed APK Wizard, First Page

If this is the first time you are going to sign a production app, you will need to create your production signing key, which you can do by clicking the "Create new..." button in the wizard. This brings up a separate dialog for describing the new signing key:

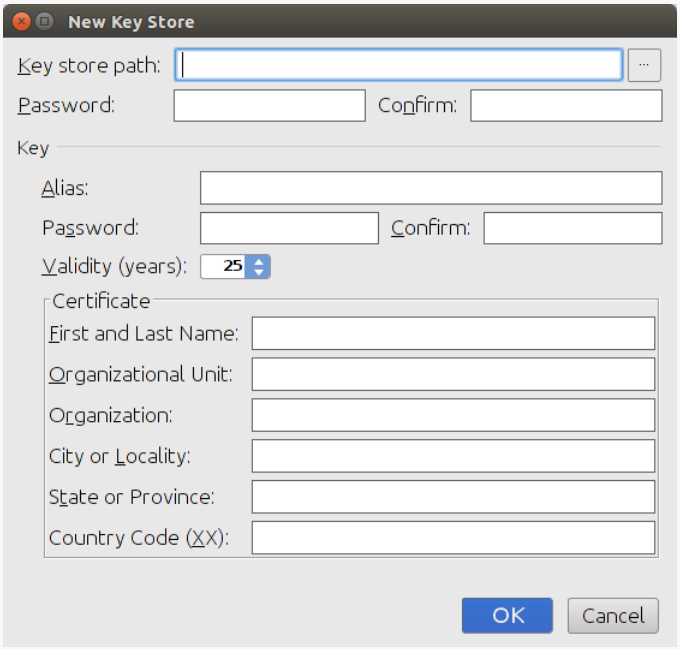


Figure 339: Android Studio New Key Store Dialog

You will need to provide a path to the keystore, manually or via the "..." button to pick a location via a dialog. You will also need to provide a password (twice) for the keystore.

You can then supply information for the signing key within the keystore, including:

- "Alias" to indicate what human-readable name we want to give the key
- "Password" and "Confirm", to specify a password for this specific key in the keystore (independent of the keystore's own password)
- "Validity", to indicate how long this key should be valid, where 25 years or more is recommended
- Details about you and your organization, asking for the standard information used in generating SSL-style keys

Clicking "OK" will generate the keystore and save it where you specified. **Be sure to back up this keystore** and record the passwords that you used.

If you already have a keystore, though, back on the first page of the "Generate Signed APK" wizard, you can click "Choose existing" to bring up a file-open dialog where you can choose your keystore. Then, fill in the keystore password, the key alias, and the key password in the dialog.

Clicking Next in the wizard brings up a page allowing you to determine what will be generated:

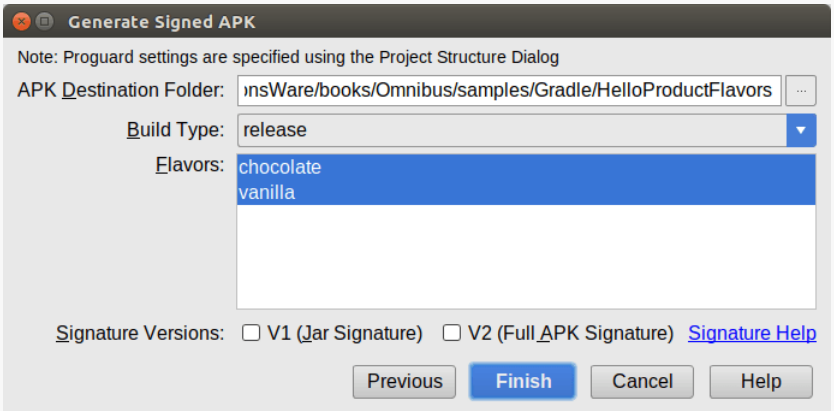


Figure 340: Android Studio Generate Signed APK Wizard, Second Page

You can indicate where the APK file should be written, what build type to use (`release` being the default), and which product flavors to use (where you can select one or several).

You can also choose which signature versions that you want to use. You have two options:

1. V1, which is the way APKs have been signed since Android 1.0
2. V2, which is an improved signature format, offering stronger protection and faster app installs, but only works on Android 7.0+

Ideally, check both signature versions. If for some reason the V2 signature format causes build problems, uncheck that version and only use V1.

Clicking "Finish" will have Android Studio begin generating the APK files. This may take some time. When it is done, a dialog will appear indicating that the work is completed. In the directory that you specified, you will get one APK file per product flavor you chose, plus manifest merger reports for those APK files. And, of course, the APK files will be signed with your chosen keystore and signing key.

Gradle

The Android Gradle Plugin can also be used to sign a production app. Curiously, this is completely independent of the mechanism that Android Studio uses to sign a

production app. Filling in the dialogs in Android Studio does not affect your `build.gradle` file, and Android Studio's "Generate Signed APK" completely ignores any manual signing configuration that you may set up in `build.gradle` (and is discussed in this section). What is covered in this section focuses on automating the signing process, to be done via a build server or just running a Gradle task from the command line.

To be able to use the Android Gradle Plugin to sign your production app, you need to provide a signing configuration to the `release` build type:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
}

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        versionCode 1
        versionName "1.0"
        minSdkVersion 15
        targetSdkVersion 25
    }

    signingConfigs {
        release {
            storeFile file('DebugActivity.keystore')
            keyAlias 'HelloConfig'
            storePassword 'laser.yams.heady.testy'
            keyPassword 'fw.stabs.steady.wool'
        }
    }

    buildTypes {
        debug {
            applicationIdSuffix ".d"
        }

        release {
            signingConfig signingConfigs.release
        }
```

```
    }
}
```

(from Diagnostics/Activity/app/build.gradle)

Here, our `release` build type has a `signingConfig` property, referencing the name of a signing configuration specified in the `signingConfigs` closure. This is used to provide rules for how to sign the APK that is assembled by Gradle. In this project's `build.gradle` file, we have a `release` closure in `signingConfigs`, supplying the requisite information about the keystore:

- The `storeFile` path, specified as a `file()` pointing to a keystore in the project's root directory
- The `keyAlias` given to the signing key inside the keystore
- The `storePassword` and `keyPassword` used to access the keystore

The `signingConfig` property in the `release` closure in `buildTypes` references the signing configuration we want as `signingConfigs.release`. All of these Groovy closures of properties in the `build.gradle` file are effectively building up a data structure, which we can access. So, `signingConfigs.release` says to find the `release` definition in the `signingConfigs` closure.

This sample bakes in the keystore data into the `build.gradle` file, including the passwords, and has the keystore in the root of the project. That is for demonstration simplicity and will not be suitable for all projects. In particular, keystores and their credentials should *not* be stored in a publicly accessible repository, as that would allow others to sign *their* apps with *your* signing key, which is not good. There are a variety of strategies for handling this, from using environment variables to requesting the data be entered on the command line, as are discussed in the chapter on advanced Gradle techniques.

Adding a `signingConfig` property in our `release` build type enables the `installRelease` task. Running `gradle tasks` will show `installRelease` as an available option, because now the Android Gradle Plugin knows how to sign the APK. Of course, there could be flaws in the signing configuration (e.g., mis-entered key alias), and that will result in build errors when you try to `installRelease` the project.

Two Types of Key Security

There are two facets to securing your production key that you need to think about:

- You need to make sure nobody steals your production keystore and its password. If somebody does, they could publish replacement versions of your applications — since they are signed with the same key, Android will assume the replacements are legitimate.
- You need to make sure you do not lose your production keystore and its password. Otherwise, even *you* will be unable to publish replacement versions of your applications.

For solo developers, the latter scenario is more probable. There already have been **many** cases where developers had to rebuild their development machine and wound up with new keys, locking themselves out from updating their own applications. As with everything involving computers, having a solid backup regimen is highly recommended. In particular, consider a secure off-site backup, such as having your production keystore on a thumb drive in a bank safe deposit box.

For teams, the former scenario may be more likely. If more than one person needs to be able to sign the application, the production keystore will need to be shared, possibly even stored in the revision control system for the project. The more people who have access to the keystore, the more likely it is somebody will wind up doing something evil with it. This is particularly true for projects with public revision control systems, such as open source projects — developers might not think of the implications of putting the production keystore out for people to access.

Distribution

It is entirely possible that the user base for your app consists solely of yourself.

However, in most cases, you are going to be giving your app to others, free or for some sort of fee.

This chapter outlines things you will need to think about when distributing your app.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on signing your app.

Get Ready To Go To Market

While being able to sign your application reliably with a production key is necessary for publishing a production application, it is not sufficient. Particularly for the Play Store, there are other things you must do, or should do, as part of getting ready to release your application.

Versioning

You need to supply `versionCode` and `versionName` values in your `build.gradle` file. The value of `versionName` is what users and prospective users will see in terms of the label associated with your application version (e.g., "1.0.1", "System V", "Loquacious Llama"). More important, though, is the value of `versionCode`, which needs to be an

integer increasing with each release — that is how Android tells whether some edition of your APK is an upgrade over what the user currently has.

Application ID

You also need to make sure that your application ID is going to be unique. If somebody tries downloading your application onto their device, and some other application is already installed with that same package name, your application will fail to install.

Your application ID defaults to be the value of your `package` attribute in your `<manifest>` element in the manifest. You can override the application ID using `applicationId` properties in `defaultConfig` or a product flavor in `build.gradle`. You can also append an `applicationIdSuffix` tied to a build type or product flavor in Gradle as well.

Since the manifest's `package` also provides the base Java package for your project, and since you hopefully named your Java packages with something based on a domain name you own or something else demonstrably unique, this should not cause a huge problem.

Also, bear in mind that your application ID must be unique across all applications on the Play Store, should you choose to distribute that way.

Icon and Label

Your `<application>` element needs to specify `android:icon` and `android:label` attributes, to supply the display name and icon that will be associated with the application in the My Applications list on the device and related screens. Your activities will inherit the icon if they do not specify icons of their own.

If you have graphic design skills, the Android developer site has guidelines for creating icons that will match other icons in the system.

Logging

In production, try to minimize unnecessary logging, particularly at low logging levels (e.g., debug). Remember that even if Android does not actually log the information, whatever processing is involved in making the `Log.d()` call will still be

done, unless you arrange to skip the processing somehow. You could outright delete the extraneous logging calls, or wrap them in an `if()` test:

```
if (BuildConfig.DEBUG) {
  Log.d(TAG, "This is what happened");
}
```

Here, `BuildConfig.DEBUG` is a `public static final boolean` value, supplied by Android, that indicates whether you are building for debug or production. Whether you adjust the definition by hand or by automating the build process is up to you. But, when `BuildConfig.DEBUG` is `false`, any work that would have been done to build up the actual `Log` invocation will be skipped, saving CPU cycles and battery life.

Conversely, error logs become even more important in production. Sometimes, you have difficulty reproducing bugs "in the lab" and only encounter them on customer devices. Being able to get stack traces from those devices could make a major difference in your ability to get the bug fixed rapidly.

First, in addition to your regular exception handlers, consider catching everything those handlers miss, notably runtime exceptions:

```
Thread.setDefaultUncaughtExceptionHandler(onBlooey);
```

This will route all uncaught exceptions to an `onBlooey` handler:

```
private Thread.UncaughtExceptionHandler onBlooey=
  new Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread thread, Throwable ex) {
    Log.e(TAG, "Uncaught exception", ex);
  }
};
```

There, you can log it, raise a dialog if appropriate, etc.

Then, offer some means to get your logs off the device and to you, via email or a Web service. Some Android analytics firms, like Flurry, offer exception stack trace collection as part of their service. There are also open source projects that support this feature, such as ACRA.

Testing

As always, testing, particularly acceptance testing, is important.

Bear in mind that the act of creating the production signed version of your application could introduce errors, such as having the wrong Google Maps V2 API key. Hence, it is important to do user-level testing of your application after you sign, not just before you sign, in case the act of signing messed things up. After all, what you are shipping to those users is the production signed edition — you do not want your users tripping over obvious flaws.

As you head towards production, also consider testing in as many distinct environments as possible, such as:

1. Trying more than one device, particularly if you can get devices with different display sizes
2. If you rely on the Internet, try your application with WiFi, with 3G, with EDGE/2G, and with the Internet unavailable
3. If you rely on GPS, try your application with GPS disabled, GPS enabled and working, and GPS enabled but not available (e.g., underground)

EULA

End-user license agreements — EULAs — are those long bits of legal prose you are supposed to read and accept before using an application, Web site, or other protected item. Whether EULAs are enforceable in your jurisdiction is between you and your qualified legal counsel to determine.

In fact, many developers, particularly of free or open source applications, specifically elect not to put a EULA in their applications, considering them annoying, pointless, or otherwise bad.

However, the Play Store developer distribution agreement has one particular clause that might steer you towards having a EULA:

> You agree that if you use the Store to distribute Products, you will protect the privacy and legal rights of users. If the users provide you with, or your Product accesses or uses, user names, passwords, or other login information or personal information, you must make the users aware that the information will be available to your Product, and you must provide legally adequate privacy notice and protection for those users. Further, your

> Product may only use that information for the limited purposes for which the user has given you permission to do so. If your Product stores personal or sensitive information provided by users, it must do so securely and only for as long as it is needed. But if the user has opted into a separate agreement with you that allows you or your Product to store or use personal or sensitive information directly related to your Product (not including other products or applications) then the terms of that separate agreement will govern your use of such information. If the user provides your Product with Google Account information, your Product may only use that information to access the user's Google Account when, and for the limited purposes for which, the user has given you permission to do so.

Hence, if you are concerned about being bound by what Google thinks appropriate privacy is, you may wish to consider a EULA just to replace their terms with your own.

Unfortunately, having a EULA on a mobile device is particularly annoying to users, because EULAs tend to be long and screens tend to be short.

Again, please seek professional legal assistance on issues regarding EULAs.

Writing a Gradle Plugin

Gradle is very extensible, from Groovy scripting code in your `build.gradle` file all the way up to dedicated Gradle plugins. The more you want to reuse a particular piece of functionality, or the more sophistication you want in your custom builds, the more likely it is that a Gradle plugin will be the right solution.

Sometimes, a Gradle plugin will be focused mostly on tailoring the build process itself. However, a Gradle plugin could be simply an integration point for a tool that could be used in other ways (e.g., command-line API) that would not otherwise require Gradle. This is particularly true for tools that work across IDEs, where you put most of the core logic into a core set of tool code and have thin build system plugins that bridge between the build system or IDE and the tool itself.

In this chapter, we will look at the basics of setting up a Gradle plugin. The plugin that we develop in this chapter will not do much of anything. The *next* chapter, however, creates a Gradle plugin that generates Java code, building on what we explore here.

Prerequisites

This chapter will make the most sense if you have read the preceding chapters on Gradle, particularly the chapter on Gradle tasks.

Customizing a Gradle Build

Gradle is Groovy — in other words, Gradle is a domain specific language (DSL) implemented in the Groovy scripting language. Hence, you are welcome to blend in Groovy code into your `build.gradle` file, for everything from simple string

concatenation to loops, branches, and everything else that you would expect from a modern OO-flavored scripting language.

You are not limited to putting that Groovy code directly in `build.gradle` either. You can use `apply from:` to indicate the path to some other Gradle script, reminiscent of a `#include` from C/C++, `<include>` elements in an Android layout resource, and so forth:

```
apply from: 'relative/path/to/some.gradle'
```

Or, you can create a `buildSrc/` directory in your project root. In there, you can have Java code (e.g., in `src/main/java/`) or Groovy code (e.g., in `src/main/groovy/`), all of which will be compiled and added to the classpath for your main `build.gradle` file. As a result, your `build.gradle` file can refer to whatever classes and such are created in this `buildSrc/` quasi-module. This is powerful, but it is inconvenient to distribute, except for other people working on the same overall project (e.g., via version control systems). As such, this approach is mostly for project-specific build system extensions.

A Gradle plugin is a separate module, akin to the `buildSrc/` directory, that creates a JAR that incorporates compiled Java and Groovy code. That plugin JAR can then be used by other modules or other projects. The plugin can be distributed through Maven Central, JCenter, or other artifact repositories. And, people can use it just like you use the Android Gradle Plugin: requesting it in a `buildscript` closure and applying the plugin. This is somewhat more complicated to set up than is a `buildSrc/` directory, but it offers the easiest path for distributing substantial build customizations.

Some Use Cases for a Custom Plugin

For lightweight tweaking of how a build is done, adding some Groovy code to your `build.gradle` file is the simplest solution. But sometimes what you want to do might get a bit involved for a smattering of lines in a `build.gradle` file, and it is these sorts of situations where a Gradle plugin becomes a useful approach.

Code Generation

Sometimes, we want to generate Java code based upon other sorts of files within our project.

The Android Gradle Plugin offers this sort of thing in a few of places:

- Our `R` class is code-generated by the `aapt` build tool, invoked through Gradle, based upon the contents of our resource directories
- The build tools similarly look for AIDL files — used in bound services — to code-generate our binding classes for us
- The data binding framework looks for layout files with a particular structure and code-generates some Java classes that implement data binding for the widgets in those layouts

Other developers do the same sort of thing. For example, Square has published the SQLDelight plugin, which code-generates a `SQLiteOpenHelper` subclass based upon a SQL script that you include in your module.

And, in the next chapter, we will see how to do this same sort of code generation.

Resource Generation

Sometimes, you want to generate other files, instead of Java. For example, Trello released the Victor Gradle plugin, which allows you to add SVG files to your module. Victor will convert those into PNGs or vector drawables for you automatically, rather than you having to do that yourself using various tools.

Code Analysis

Sometimes, we are not trying to create things, but instead are trying to analyze what is there and fail the build if the analysis turns up problems.

For example, a standard plugin distributed with Gradle integrates with the Checkstyle library to confirm that the source code adheres to coding standards.

Writing a Plugin

Creating a Gradle plugin is not especially difficult. It is somewhat esoteric and not especially well-documented, but this chapter hopefully will address those particular issues for you.

Create a Java Module

One major difference in creating a Gradle plugin versus creating an Android app is where the code runs. Android apps run on Android. Gradle plugins run on your development machine. As such, our code cannot use Android-specific APIs, so we need a module that knows about ordinary Java, not Android.

Android Studio's new-module wizard (New > New Module from the File menu) offers an option for "Java Library":

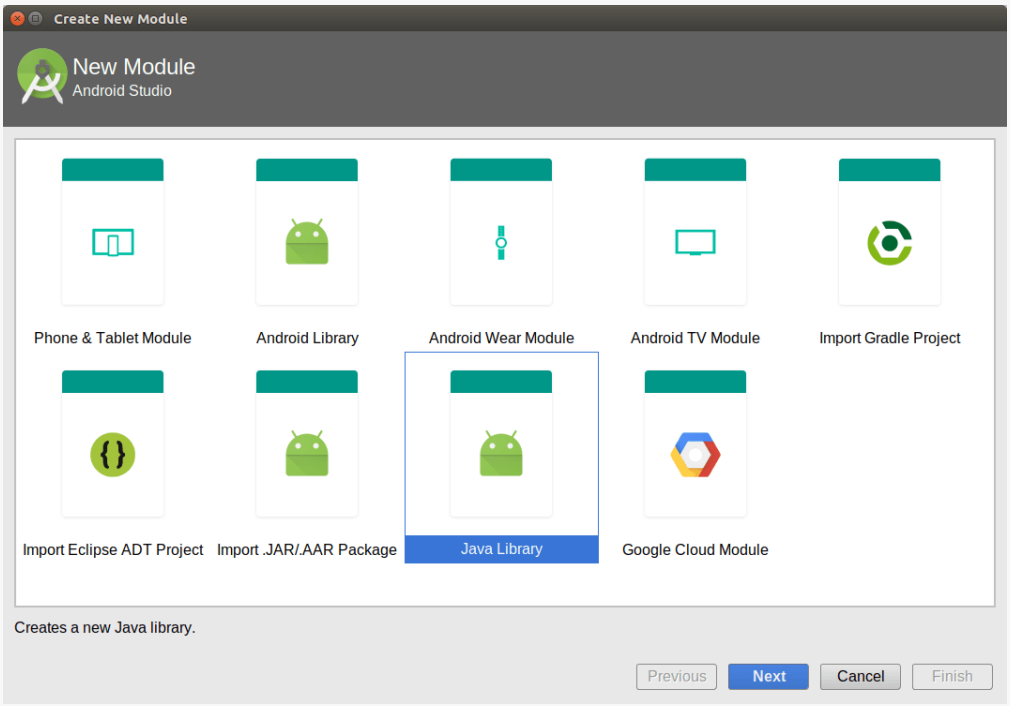


Figure 341: Android Studio New-Module Wizard, with Java Library Option

This will add a new module to the project, but one that uses the standard `java` Gradle plugin for building the code, rather than the `com.android.application` or `com.android.library` plugins used to create Android apps and libraries.

The second page of the new-module wizard asks some questions about the module to be created:

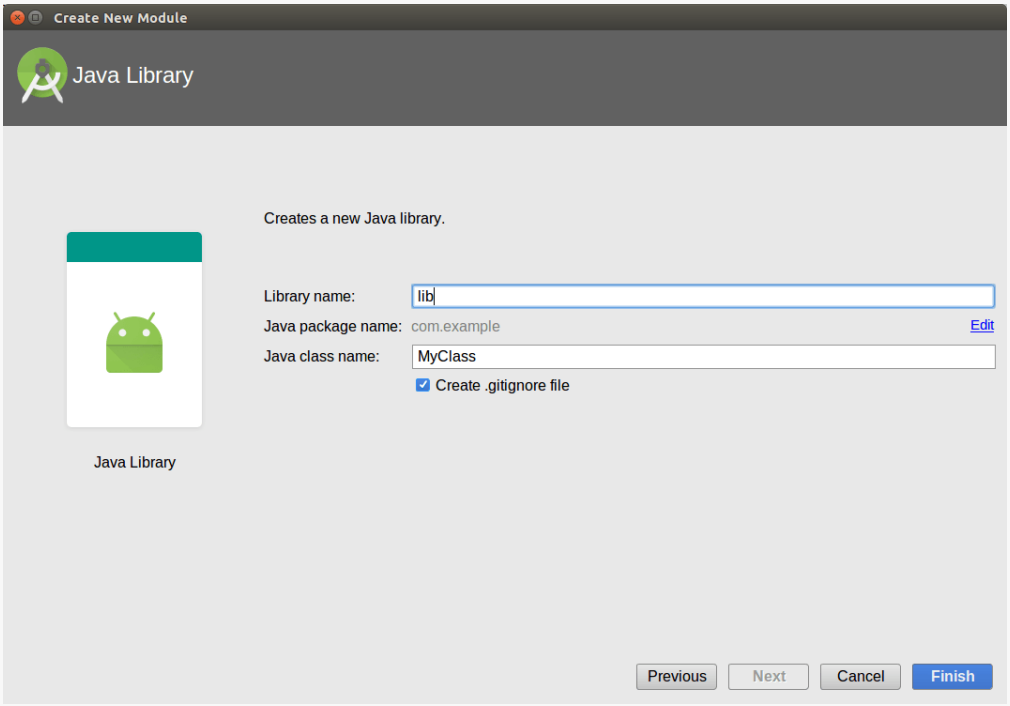


Figure 342: Android Studio New-Module Wizard, with Java Library Configuration

The big item is the "Library name", which will be the name of the module added to your project. If you are adding this to a large existing project — for example, adding a Gradle plugin to an existing tool – you might use `gradle-plugin` or something as the module name. If the project is creating a dedicated Gradle plugin, you might name the module after the plugin itself (e.g., the `staticizer` plugin that we will create in the next chapter). Or, you could take the approach used by the `Gradle/PluginStub` sample project, and call it `plugin`.

Creating a new Java library module forces you to create a Java class for use by the library. As you will see shortly, this is not a requirement for creating a Gradle plugin, though often you will wind up using Java classes along the way. You can choose the Java package name and class that makes the most sense for your code. There are no particular naming rules for Gradle plugins that you have to follow.

Apply the Groovy Plugin

The typical recipe for writing a Gradle plugin involves both Java code and Groovy code. That is not strictly required, as you could write the entire plugin in Java. For illustration purposes, the `PluginStub` sample project follows convention and uses Groovy along with Java.

However, an Android Studio-created Java module knows nothing about Groovy. It knows about Java, as the module's `build.gradle` file will have `apply plugin: 'java'` to teach Gradle that this module will contain Java code and create a Java JAR as output.

If you want to also use Groovy code, simply add `apply plugin: 'groovy'` to the module's `build.gradle` file:

```
apply plugin: 'groovy'
apply plugin: 'java'
```

(from Gradle/PluginStub/plugin/build.gradle)

Update the Dependencies

The plugin's module will also need some new dependencies:

```
dependencies {
    implementation gradleApi()
    implementation localGroovy()
}
```

(from Gradle/PluginStub/plugin/build.gradle)

The `gradleApi()` dependency pulls in Gradle's own internal API, for use by plugins. The `localGroovy()` dependency says that we can add a `src/main/groovy/` directory and put Groovy-defined Java classes in there, and they too will be added to the project.

Add a Groovy Source Directory and Package

Java classes defined in Groovy go into the same sort of Java package directory structure that regular Java classes do. It just so happens that they will reside under a `groovy/` directory, rather than a `java/` directory.

The `plugin/` module of the sample project has a `src/main/groovy/com/commonsware/android/gradle/plugin/` directory:

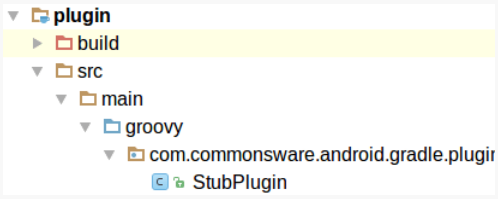


Figure 343: Android Studio, Showing Groovy Directory Structure

In there is a `StubPlugin.groovy` file containing some Groovy code.

Implement a Plugin

That code comes in the form of a `StubPlugin` class:

```
package com.commonsware.android.gradle.plugin

import org.gradle.api.Plugin
import org.gradle.api.Project

public class StubPlugin implements Plugin<Project> {
  @Override
  public void apply(Project target) {
    def myTask = target.tasks.create("stubTask") << {
      def helper = new StubHelper();
      println helper.getMessage();
    }

    myTask.group = "commonsware"
    myTask.description = "Do something useful"
  }
}
```

(from Gradle/PluginStub/plugin/src/main/groovy/com/commonsware/android/gradle/plugin/StubPlugin.groovy)

Again, this class could have been written in Java — there is nothing specific about plugin classes that requires Groovy. Groovy syntax for class definitions closely resembles that of Java.

Here, `StubPlugin` implements a `Plugin`, one creating a `Project`. Gradle plugins could be applied to other sorts of Gradle artifacts, at least in principle, though most samples you will see involve a `Project`. Here, `Project` refers to what Android Studio calls a module, and what Gradle itself refers to as a sub-project. In other words, the `Project` is a model object describing the actual app code that is being built, where our plugin can now interact with the build process.

The entry point into a `Plugin` is `apply()`, where you are given the build artifact (e.g., the `Project`), and your job is to tinker with the build process.

Gradle scripts are recipes for building an object model that, in turn, describes how to build a project. Similarly, what our plugin needs to do is *modify that recipe*, adding in hooks that will eventually get run during actual builds, where we can do special stuff.

A typical way of implementing this is to create one or more additional tasks, just like the tasks that Gradle itself has, or that the Android Plugin for Gradle adds. Here, we have Groovy code that defines a new task, named `stubTask`, being added to the roster of tasks for this `Project`. The Groovy code between the braces represents the work to be done by this Gradle task, and we will return to that code shortly.

The rest of `apply()` configures the rest of the task, putting it into a specific group (for use by Android Studio or other IDEs) and adding a description (for use with tooltips, generated documentation, and the like).

Using Both Groovy and Java

If you elect to use some amount of Groovy code with your plugin, the question then becomes: do you need Java at all, and if so, when?

In principle, Groovy can do everything that Java can. In practice, how much Groovy you want to write will depend largely on your comfort level with Groovy and with multi-language programming. You might elect to designate some in-app boundary as also serving as the boundary between programming languages. For example, you might say that plugin-specific code is written in Groovy, but code that is usable via other interfaces gets written in Java.

`StubPlugin` demonstrates integrating with ordinary Java code by means of the `StubHelper` instance that it creates and uses to retrieve a message (`getMessage()`), which `apply()` then dumps to the console via `println`. `StubHelper` is an ordinary Java class, in the `java/` peer set of directories to the `groovy/` directories. In particular, `StubHelper` happens to be in the same Java package as is `StubPlugin`, so we do not need an `import` statement for `StubHelper`. `StubHelper` itself is trivial, returning a static `String` as the result of `getMessage()`:

```
package com.commonsware.android.gradle.plugin;

public class StubHelper {
```

```
  String getMessage() {
    return("Hello, world!");
  }
}
```

(from Gradle/PluginStub/plugin/src/main/java/com/commonsware/android/gradle/plugin/StubHelper.java)

Adding Plugin Metadata

In an Android app, merely having Java classes inheriting from "magic" base classes like `Activity` or `Service` is insufficient. We also need to have some metadata to teach Android about our activities, services, and so on. In the case of an Android app, that metadata comes in the form of the manifest.

Similarly, we need to teach Gradle that:

- Our library contains one or more Gradle plugins
- What those Gradle plugins are called
- What Java/Groovy class is the entry point for the plugin

That is handled by its own form of metadata.

This metadata is held in a `resources/` directory inside of the `main/` source set. Here, `resources/` refers to classic Java resources, otherwise known as "semi-random files that wind up being stuffed into the JAR". It has little to do with *Android* resources that you might put in a `res/` directory in an Android project.

A typical convention for JARs is to have a `META-INF/` folder containing metadata. That is set up in a Java module by having a `resources/META-INF/` directory.

Gradle, in turn, wants to see a `gradle-plugins/` directory inside of `META-INF/`, with one or more `.properties` files. The base name of the `.properties` file (i.e., the name without the file extension) is the full name of the plugin. Convention nowadays is for third-party plugins to use Java-style reverse-domain-name names, like `com.android.application` and `com.android.library`. So, inside of `resources/META-INF/gradle-plugins/`, our Java module has `com.commonsware.android.gradle.plugin.properties`, to define a `com.commonsware.android.gradle.plugin` plugin.

The properties file needs only one line, defining the value for an `implementation-class` property. The value is the fully-qualified class name of the `Plugin` subclass that is the entry point for the plugin:

```
implementation-class=com.commonsware.android.gradle.plugin.StubPlugin
```

(from Gradle/PluginStub/plugin/src/main/resources/META-INF/gradle-plugins/com.commonsware.android.gradle.plugin.properties)

Distributing the Plugin

When you build the `plugin/` module, the result is a `plugin` JAR file located in `build/libs/` of the module. That JAR is what modules and projects that want to use the plugin need to access.

You could pass around the bare JAR file. Nowadays, the preferred approach for distributing JARs is to do so via artifacts in repositories, the way that we use such artifacts for libraries that we want to add to our Android apps. In fact, we have already seen one such plugin distributed via an artifact: the `com.android.tools.build:gradle` artifact that we reference in the `buildscript` `dependencies` closure in our top-level `build.gradle` file.

There are formal ways to publish artifacts to places like JCenter or Maven Central. For lightweight testing purposes, you can easily set up a Maven-style repository on your own development machine, as this sample app does. And there are "middle-ground" options, such as having a repository on an internal server somewhere for use by development team within an organization.

The `build.gradle` file for the `plugin/` module sets up tasks, via the `maven` plugin, to deploy the plugin JAR as an artifact to a repository located in a `repository/` directory off of the project root. This would be reasonable for a plugin that is being used by other modules in that project but not other projects, as is the case here.

```
apply plugin: 'groovy'
apply plugin: 'java'

dependencies {
    implementation gradleApi()
    implementation localGroovy()
}

sourceCompatibility = "1.7"
targetCompatibility = "1.7"

group 'com.commonsware.android.gradle'
version '0.0.1'

apply plugin: 'maven'
```

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: uri('../repository'))
        }
    }
}
```

(from Gradle/PluginStub/plugin/build.gradle)

The `group` provides the artifact group ID for this plugin — the name that appears left-most in the `group-id:artifact-id:version` triple that we use when referencing these sorts of dependencies. The artifact ID is the name of the module itself by default.

The `uploadArchives` configuration teaches the `maven` plugin where we want to push the artifact to. In this case, it is to a `../repository/` directory relative to the module root, which places the `repository/` directory off of the project root.

You can run the `uploadArchives` task, directly from Gradle if you wish, by opening the Gradle tool (typically docked on the right) and finding the `uploadArchives` task within an `upload` task group in the `:plugin` module:

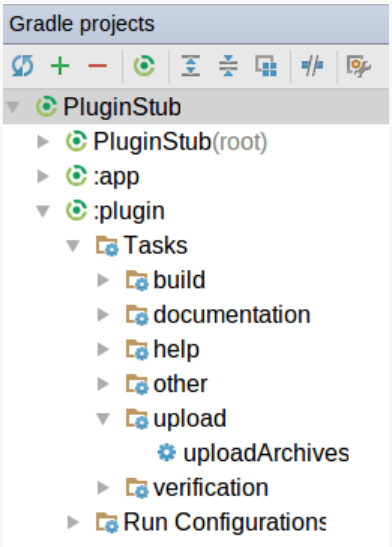


Figure 344: Gradle Task Tree, Showing uploadArchives Task

Once you run that task, you will find a `repository/` directory off of the project root, with our JAR file as an artifact:

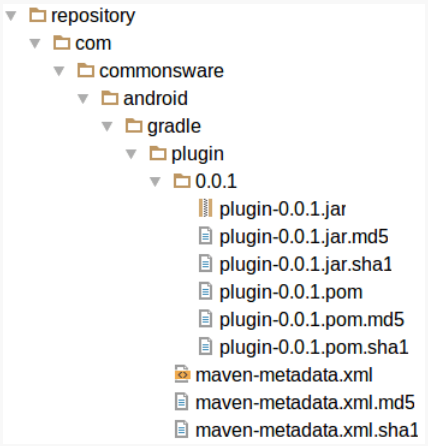


Figure 345: Gradle Task Tree, Showing Local Repository

Using the Plugin

Now, other modules in this project can use the plugin, such as the `app/` module, containing an Android app created through the Android Studio new-project wizard, without any changes to the generated Java code, resources, or manifest.

In a typical Android Studio project, you see plugins added via the `buildscript` closure in the `build.gradle` file in the project's root directory. Technically, this is only required for plugins that either all modules will use, or enough modules will use that it makes sense to define it in one place.

A module's own `build.gradle` file can also load third-party plugins, using the same `buildscript` closure. This will be added to the plugins loaded from the project's `build.gradle` file to create the complete list of possible plugins for the module.

So, the `app/` module takes this approach in its `build.gradle` file:

```
buildscript {
    repositories {
        maven {
            url uri('../repository')
        }
    }
    dependencies {
        classpath 'com.commonsware.android.gradle:plugin:0.0.1'
    }
}
```

```
apply plugin: 'com.android.application'
apply plugin: 'com.commonsware.android.gradle.plugin'
```

(from Gradle/PluginStub/app/build.gradle)

Here, we indicate that the plugin search path should include our custom `repository/`, and that we are looking for a plugin artifact identified as `com.commonsware.android.gradle:plugin:0.0.1`. That lines up with the group ID, artifact ID, and version number that our plugin used, and this JAR can be found in our local repository, assuming that we have run the `uploadArchives` tasks from the `plugin/` module.

We also use `apply plugin: 'com.commonsware.android.gradle.plugin'` to actually use the plugin. Here, the name of the plugin comes from the name of the `.properties` file that we put in the `resources/META-INF/gradle-plugins/` directory of the `plugin/` module.

The rest of the `app/` module's `build.gradle` file (not shown) is the standard stuff that you get in a `build.gradle` file for an application module out of the new-project wizard.

With that in place, we now have a new `stubTask` task, inside of a `commonsware` group (per our plugin's task configuration in the Groovy code) that we can run:

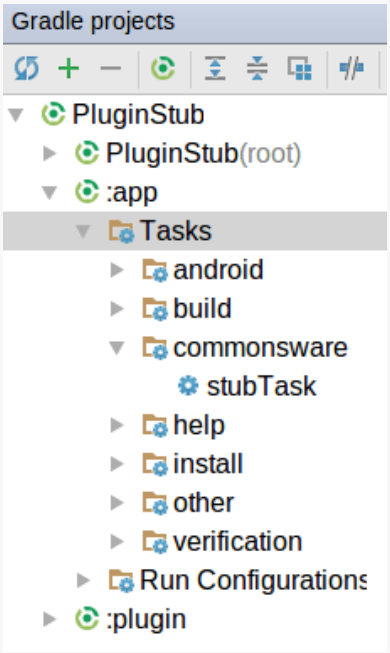


Figure 346: Gradle Task Tree, Showing stubTask Task

If you run that task — and if Android Studio is behaving today – you will see our static message show up in the Gradle Console output:

```
Executing tasks: [stubTask]

Configuration on demand is an incubating feature.
Observed package id 'system-images;android-14;default;armeabi-v7a' in inconsistent
location '/opt/android-sdk-linux/system-images/android-14/armeabi-v7a' (Expected
'/opt/android-sdk-linux/system-images/android-14/default/armeabi-v7a')
Observed package id 'system-images;android-14;default;armeabi-v7a' in inconsistent
location '/opt/android-sdk-linux/system-images/android-14/armeabi-v7a' (Expected
'/opt/android-sdk-linux/system-images/android-14/default/armeabi-v7a')
Incremental java compilation is an incubating feature.
:app:stubTask
Hello, world!

BUILD SUCCESSFUL

Total time: 13.707 secs
```

Dealing with Bugs and Changes

If you develop your own Gradle plugin, you may find the development process to be a bit of a pain. This is particularly true in cases like the sample project, where the plugin and a module using the plugin exist in the same project.

For example, what happens when you run `uploadArchives` and the plugin has a bug?

In this particular case, usually there is not much of a problem, insofar as nothing depends upon `stubTask`. However, if your bug affects the plugin as a whole — for example, you screw up the `.properties` file – now your entire build process is broken, *even for the plugin module*. In the case of a plugin configuration bug, you cannot successfully sync your project with the Gradle build files.

When you get into more sophisticated plugins, where you chain new tasks onto the existing build process (e.g., for code generation), you will run into cases where bugs in the plugin allow you to still sync your project with the Gradle build files, but attempts to build anything trip over the plugin bug. This can even affect builds of the *plugin*, depending on the nature of the plugin bugs.

As a result, you will find yourself commenting out the use of the plugin in the other module(s) from time to time, cleaning the project, fixing the plugin bug, running `uploadArchives` with the repaired plugin, then uncommenting those commented-out lines, and trying it again. Depending on how good you are with bug-fixing, this may take some time.

Also, even though the plugin is being used by one module and the implementation is in a separate module, Android Studio may erroneously flag your Groovy-defined classes as having errors, indicating that these classes already exist. Just clean the project, and those spurious errors should go away.

Trying the Sample App

The sample app is set up to apply the custom plugin. Unfortunately, when you first check out this project, that custom plugin does not yet exist. And, due to the way Android Studio and Gradle work, you cannot build the plugin, because the `app/` module fails the build due to the missing plugin... even if you are not trying to build the `app/` module.

So, to try out this sample app, you will need to:

- Comment out the `buildscript` closure and the `apply plugin:` statement in `app/build.gradle` that refer to our custom plugin
- Import the project into Android Studio
- Run the `uploadArchives` tasks in the `plugin/` module, to build and "upload" the plugin to our local repository
- Uncomment what you commented in the first bullet above

Then, everything should be set.

Creating a Real Plugin

The stub plugin shown in this chapter is extremely rudimentary. It is enough to demonstrate the scaffolding necessary to create a plugin, and little else.

The `Staticizer` plugin demonstrated in the next chapter is a more sophisticated sample, generating Java code to be blended into a project build, just like the data binding framework, SQLDelight, and other Gradle plugins do.

But, `Staticizer` needs a few more features in its plugin than `PluginStub` needed.

Integrating with the Android Plugin

If you want to add tasks that tie into the build process of an Android app or library, you will wind up needing to reference portions of the public API exposed by the Android Gradle Plugin. That, in turn, will require you to have the plugin depend upon the Android Plugin for Gradle, so you can reference the appropriate symbols:

So, you might have a `dependencies` closure in the plugin's `build.gradle` file like this:

```
dependencies {
    compile gradleApi()
    compile localGroovy()
    implementation 'com.android.tools.build:gradle:2.2.2'
}
```

Here, we are saying that we are dependent upon the Android Plugin for Gradle (specifically version 2.2.2).

You can use similar `implementation` statements to pull in other libraries that your plugin might need, such as Gson.

Validating the Environment

If you are planning on integrating into the Android build process, you may want to validate that your plugin is being used in a module that has *also* applied the `com.android.application` or `com.android.library` plugin. Otherwise, your plugin would have nothing to do.

Chiu-ki Chan demonstrated an approach for doing this in her `gce2retrofit` Gradle plugin example. In her case, her plugin can work with either of the two Android plugins or the plain `java` plugin. She takes advantage of the fact that a `Project` knows its `plugins`, and we can use `hasPlugin` to see if the `Project` is configured with certain plugins or not:

```
package com.sqisland.gce2retrofit;

import com.android.build.gradle.AppPlugin;
import com.android.build.gradle.LibraryPlugin;

import org.gradle.api.Project;
import org.gradle.api.Plugin;
import org.gradle.api.plugins.JavaPlugin;

public class GradlePlugin implements Plugin<Project> {
  @Override
  public void apply(Project project) {
    def hasAndroidApp = project.plugins.hasPlugin AppPlugin
    def hasAndroidLib = project.plugins.hasPlugin LibraryPlugin
    def hasJava = project.plugins.hasPlugin JavaPlugin
    if (!hasAndroidApp && !hasAndroidLib && !hasJava) {
      throw new IllegalStateException("'android' or 'android-library' or 'java'
plugin required.")
    }

    // more code here
```

(from her plugin's `GradlePlugin.groovy` script)

Here, she checks for all three possible plugins and throws an exception if they are all missing.

Writing a Task Class

In the `PluginStub` sample, our task was implemented as the Groovy equivalent of a Java anonymous inner class. For trivial tasks, that is a reasonable approach. The

bigger your task, the more likely it is that you will want to create a formal dedicated class for the task.

In that case, rather than using syntax like this to add a task:

```
def myTask = target.tasks.create("stubTask") << {
  def helper = new StubHelper();
  println helper.getMessage();
}

myTask.group = "commonsware"
```

you use syntax like this:

```
def myTask = project.tasks.create("commonsware", SomeTask)
```

Here, `SomeTask` is a dedicated class, extending Gradle's `DefaultTask` base class. That class can be written in Java or Groovy, much as the plugin class itself can be written in Java or Groovy.

Splicing Your Task Into the Build

In the `PluginStub` sample, `stubTask` is a standalone task. Running other tasks, such as building an app, would never invoke `stubTask`. Occasionally, this is what you want. Other times, you want your custom task to be invoked as part of the overall build process. For example, with code generation, you want to get a chance to generate Java code at the same points in time when other Java code gets generated (e.g., `R` class), before the Java code gets compiled.

That gets to be a bit complicated. While Gradle has a fairly simple approach for declaring that one task depends upon another, in an Android build, we have many build variants to deal with, based on build type and, possibly, product flavor.

The recipe is to create a custom task for *each* build variant:

```
variants.all { variant ->
  def task=
      target.tasks.create("doSomething${variant.name.capitalize()}",
          DoSomethingTask)
  variant.javaCompile.dependsOn task
}
```

Here, we create a custom task, for a `Project` named `target`. The task is named `doSomething...`, where the `...` is replaced by the name of the build variant with the first letter capitalized. So, we wind up with custom tasks like `doSomethingDebug` or `doSomethingAmazonRelease` or whatever.

We also indicate that the existing task that performs the Java compilation (`variant.javaCompile`) depends upon our new task. Hence, our task will be executed before the Java code gets compiled.

And, we do this in a loop over all possible build variants (`variants.all`) for whatever module our code is in. The catch is that getting that list of variants varies, depending upon whether we are building an Android application or an Android library:

```
def isApp=target.plugins.hasPlugin AppPlugin
def isLib=target.plugins.hasPlugin LibraryPlugin

if (!isApp && !isLib) {
  throw new
    IllegalStateException("This plugin depends upon the com.android.application or
com.android.library plugins")
}

def variants

if (isApp) {
  variants=target.android.applicationVariants
}
else {
  variants=target.android.libraryVariants
}
```

We set `variants` to be the `applicationVariants` or the `libraryVariants` depending upon whether we are in an Android app module or an Android library module.

Configuring the Plugin

When we write our Android apps and libraries, we almost always have an `android` closure that we use to configure the behavior of the Android build process and results (e.g., `minSdkVersion`).

This is what Gradle refers to as an “extension object”, and your plugin can register one of these.

Simply create a Java or Groovy class that defines what properties you want to be configurable:

```
class StaticizerConfig {
  def String packageName
}
```

Then, create a new extension object, by calling `create()` on the `extensions` collection inside of the `Project` (here named `target`):

```
target.extensions.create('staticizer', StaticizerConfig)
```

This will allow users of this plugin to create a `staticizer` and supply a value for a `packageName` property:

```
apply plugin: 'com.commonsware.android.staticizer'

staticizer {
    packageName 'com.commonsware.android.staticizer.demo'
}
```

We will see more about this `staticizer` extension object in the next chapter.

Code Generation

One common thing to do in a Gradle plugin is to generate Java code. Code that writes other code based upon supplied inputs is referred to as a "code generator".

At its core, Java code is just text. Developers generate text on the fly quite a bit, particularly in Web development. There are two main approaches to such text generation:

1. Use templates, when the text is largely fixed but has some values that need to be replaced at runtime
2. Use regular code to generate the text, perhaps using APIs that facilitate writing out text formats with as little coding overhead as possible

In this chapter, we will examine what it takes to generate Java code from a Gradle plugin. In particular, we will look at the second approach, where we will have Java code that generates other Java code, using a library that makes this a lot easier than writing a lot of `append()` calls to a `StringBuilder`.

Prerequisites

Understanding this chapter requires that you have read the preceding chapter and all of its prerequisites.

What Drives the Custom Code?

You could generate code based off of random numbers. More likely, there is some sort of input that you are using to determine the code to be generated. This input

can be divided roughly into three areas: other Java code, other project files, and everything else.

Other Java Code

Modern Android code gets littered with annotations. Some are from standard Java (e.g., `@Override`). Some are from Android libraries (e.g., `@NotNull`). Some are from third-party libraries, like greenrobot's EventBus (e.g., `@Subscribe`).

Sometimes, these annotations are for validation at compile time. `@Override` tells Java compilers that we think that we are overriding a method from a superclass or are implementing an interface method, and so the compile should fail if our method signature does not match anything that we could be overriding. `@NotNull` indicates that a parameter should not be `null`, allowing static code analysis to help point out places where we might accidentally pass `null` in as a value.

Sometimes, these annotations are used by third-party libraries at runtime. greenrobot's EventBus, for example, will look for `@Subscribe` annotations to determine how to deliver events to registered event-handling objects.

Sometimes, these runtime annotations generate something resembling code at runtime. Retrofit, for example, creates an instance of our service interface at runtime, with an implementation that will make the Web service requests that we desire. However, this Java bytecode is generated on devices, not as ordinary Java source files that are included in the build.

But, sometimes, these annotations get used by compile-time annotation processors. These add-ons to the build process read in Java code, analyze it (annotations in particular), and code generate Java code in support of our existing Java code. Google's AutoValue processor, for example, looks for `@AutoValue`-annotated `abstract` classes and code-generates a concrete implementation of a "value class" (one with immutable members), complete with `hashCode()`, `equals()`, and related methods.

Other Project Files

A code generator might generate Java code from other types of input that are in the project:

- The data binding framework looks for layout resources with a specific structure (root `<layout>` element) and code-generates Java binding classes for them
- SQLDelight code-generates a `SQLiteOpenHelper` subclass based upon a SQL script that you include in your module

Here, the inputs are resources or other files, not annotated Java code.

Other Data Sources

There is nothing stopping you from generating code based on inputs that come from outside of the project. For example, whereas SQLDelight works off of SQL scripts in your project tree, you can imagine a similar code generator that retrieved schemas from a live database using database I/O.

The big advantage of limiting code generation to files in the project is having reproducible builds based on version control. Ideally, years from now, you should be able to check out a branch or tag from a version control system and be able to build the project the same way then as you do on the day you committed that code to version control. That works best when the entire project specification is included in version control. A server is not, and the server response in a few years might differ from what the server response is today.

One workaround for this is to have two tools. One retrieves the data from the server and writes that data out into files that go into your project (e.g., JSON files). You would run this code from time to time, when you specifically need to get the latest configuration from the server. However, the second tool performs the code generation, working off of the project files, not from the live server data. That way, you can perform that code generation again in the future in a reproducible fashion.

Java as Poetry

In principle, all you need is `StringBuilder` to build up a Java source file to eventually write to disk. Similarly, in principle, all you need is `StringBuilder` to construct a Web page for your Web app to serve to the browser. In both cases, there are better solutions that can simplify the work and make it less prone to bugs.

In the case of Java code generation, the leading solution is JavaPoet, yet another library from Square. JavaPoet offers a fluent, builder-style API for defining the pieces

of a Java source file, such as fields and methods wrapped in classes. Then, JavaPoet can write out a formatted Java source file based upon your supplied specification.

JavaPoet is based on Square's earlier JavaWriter library, which has been discontinued.

The code-generating sample app described in this chapter takes advantage of JavaPoet for generating its Java code.

Writing a Code Generation Plugin

Sometimes, your app has some pure data that it needs. For example, the APK edition of this book needs to know what chapters there are, where the chapter HTML files are, what the chapter titles are, and so forth.

A typical approach for handling this is to have some data file that you package in your app. Perhaps it is an XML resource (`res/xml/`), a raw resource (`res/raw/`), or an asset (`assets/`). You then have code that reads in the data at runtime and uses it.

This has a couple of downsides:

- Your code gets littered with checked exceptions, for I/O errors or parsing errors that, in principle, should never occur with pre-packaged validated data files
- This sort of disk I/O should be done on a background thread, making it more painful to get this data into your app

The `Gradle/Staticizer` sample project implements a Gradle plugin that takes some static JSON and code generates a Java class exposing that data. Rather than having to read in and parse the JSON at runtime, you just refer to the generated Java class. This eliminates the extraneous exceptions and reduces the effective cost for loading the data off of disk (as we do not worry about somehow loading Java code in the background).

This particular sample is not very sophisticated. It is limited to JSON files describing objects made up of primitives (numbers and strings), such as:

```
{
  "foo": "foobar",
  "versionCode": 1
}
```

(from Gradle/Staticizer/app/src/main/staticizer/TestData.json)

A production-grade implementation would handle nested Java objects, arrays, and the like. But, this demonstrates the basic concept of code generation, in a not-unrealistic scenario, just with a limited implementation to keep the sample simple.

Designing the Output

We want a Java class that offers up the same data that is in the JSON, just without the runtime JSON loading and parsing. For example, the `TestData.json` file shown above, represented in Java, might look like:

```
package com.commonsware.android.staticizer.demo;

import java.lang.String;

public final class TestData {
  public static final String FOO = "foobar";

  public static final double VERSION_CODE = 1.0;
}
```

(from Gradle/Staticizer/app/build/generated/source/staticizer/debug/com/commonsware/android/staticizer/demo/TestData.java)

Early on, you might hand-craft such a Java class, to test out how it works and ensure that you have the right fields, data types, and so on for your needs. Then, once you have the class designed the way you want, you can determine how to map the data from your input into the desired output.

In the case of this sample app, we take a fairly literal approach to converting the JSON to Java:

- The class name is based on the JSON file name (`TestData.json` turns into `TestData.java`)
- The package should be configurable, as part of setting up the plugin
- The field names are based on the object keys from the JSON, but converted into `ALL_CAPS_WITH_UNDERSCORES` (a typical Java convention for `static` field names) instead of `camelCase` (a typical JSON convention for object attribute names)
- The fields are `public static final`, akin to the fields on the code-generated `R` class that you get from the Android build tools

- The data types should roughly match the JSON input, so if the JSON has a number, we emit an appropriate numeric data type (e.g., `double`), otherwise we emit a `String`

Crafting the Plugin

The plugin itself is based upon the `PluginStub` sample project from the preceding chapter. In this case, the plugin is in the `staticizer/` module, which once again is a plain Java module, not an Android application or library module.

The `build.gradle` file for the `staticizer/` module is similar to the one from the `PluginStub` sample:

```
apply plugin: 'groovy'
apply plugin: 'java'

dependencies {
    compile gradleApi()
    compile localGroovy()
    compile 'com.android.tools.build:gradle:2.2.2'
    compile 'com.squareup:javapoet:1.7.0'
    compile 'com.google.code.gson:gson:2.7'
    compile 'com.google.guava:guava:19.0'
}

sourceCompatibility = "1.7"
targetCompatibility = "1.7"

group 'com.commonsware.android'
version '0.0.1'

apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: uri('../repository'))
        }
    }
}
```

(from Gradle/Staticizer/staticizer/build.gradle)

It has four new dependencies:

- the Android Gradle Plugin, as we will be hooking into Android-specific build tasks
- JavaPoet, for our Java code generation
- Gson, for parsing the JSON
- Google's Guava library, for some utility classes, specifically for handling the case conversion of our field names

(Guava has a bit of a bad reputation in Android circles owing to the large size of the library, but bear in mind that this is a Java module creating a Gradle plugin, so the Guava code is not running on an Android device)

We have a `com.commonsware.android.staticizer.properties` file in `main/resources/META-INF/gradle-plugins/`, pointing to a `com.commonsware.android.staticizer.StaticizerPlugin` class:

```
implementation-class=com.commonsware.android.staticizer.StaticizerPlugin
```

(from Gradle/Staticizer/staticizer/src/main/resources/META-INF/gradle-plugins/com.commonsware.android.staticizer.properties)

The `StaticizerPlugin` class is implemented in Groovy:

```
package com.commonsware.android.staticizer

import org.gradle.api.Plugin
import org.gradle.api.Project
import com.android.build.gradle.AppPlugin
import com.android.build.gradle.LibraryPlugin

public class StaticizerPlugin implements Plugin<Project> {
  @Override
  public void apply(Project target) {
    def isApp=target.plugins.hasPlugin AppPlugin
    def isLib=target.plugins.hasPlugin LibraryPlugin

    if (!isApp && !isLib) {
      throw new
        IllegalStateException("This plugin depends upon the com.android.application or com.android.library
plugins")
    }

    target.extensions.create('staticizer', StaticizerConfig)

    def variants

    if (isApp) {
      variants=target.android.applicationVariants
    }
    else {
      variants=target.android.libraryVariants
    }
```

```
    variants.all { variant ->
      def task=
          target.tasks.create("staticize${variant.name.capitalize()}",
              StaticizerTask)

      task.outputDir=
          new File("${target.buildDir}/generated/source/staticizer/${variant.name}")
      task.group="commonsware"
      task.description="Generate ${variant.name} Java code from JSON"

      variant.javaCompile.dependsOn task
      variant.registerJavaGeneratingTask task, task.outputDir
    }
  }
}

class StaticizerConfig {
  def String packageName
}
```

(from Gradle/Staticizer/staticizer/src/main/groovy/com/commonsware/android/staticizer/StaticizerPlugin.groovy)

Here, we:

- Validate that the module also has the `com.android.application` or `com.android.library` plugins
- Create a `staticizer` extension object, using the `StaticizerConfig` class, to allow a module using this plugin to give us a `packageName` to use for generating the Java class files
- Get the build variants for the library or application
- Iterate over those variants and create a custom task for each, using a `StaticizerTask` class that we will see shortly

Each task is given a name based on the build variant name (e.g., `staticizerDebug`), courtesy of Groovy's string interpolation (`${}` syntax), the `name` property of the build variant, and the `capitalize()` method to capitalize the first letter.

We need to tell the task where it should write the Java code to. If you look in `build/generated/source/` for a typical Android module, you will see a bunch of directories of generated source files. The top directory will be an indication of what the source pertains to (e.g., `r` for the `R` class, `buildConfig` for the `BuildConfig` class). In there, you will see subdirectories by build variant (e.g., `buildConfig/debug/`), and in there will be a typical Java package directory tree leading to source code. So, we follow the same basic pattern:

- Get the `build/` directory via `target.buildDir`
- Append `generated/source/` to get to the standard location for generated source code

- Append `staticizer/` to give us a hopefully-unique directory of source
- Append the variant name so that each variant's edition of this task has its own output directory

We also:

- Associate the task with a `commonsware` group
- Give the task a description, blending in the build variant name for fun
- Say that the standard Java compilation task depends upon this new task, so our task will run before the Java code gets compiled
- Indicate that our task is generating Java source code (`registerJavaGeneratingTask`)

Writing the Task

`StaticizerTask` is a separate Groovy-defined class, extending `DefaultTask`, that serves as the task implementation:

```
package com.commonsware.android.staticizer

import org.gradle.api.DefaultTask
import org.gradle.api.tasks.InputDirectory
import org.gradle.api.tasks.OutputDirectory
import org.gradle.api.tasks.TaskAction
import org.gradle.api.tasks.incremental.IncrementalTaskInputs

public class StaticizerTask extends DefaultTask {
  @InputDirectory
  File inputDir=new File(project.getProjectDir(), "src/main/staticizer")

  @OutputDirectory
  File outputDir

  @TaskAction
  public void execute(IncrementalTaskInputs inputs) {
    if (!project.staticizer.packageName) {
      throw new IllegalStateException('staticizer.packageName is undefined!')
    }

    def staticizer=new Staticizer();

    for (File input : inputDir.listFiles()) {
      if (!input.name.startsWith(".")) {
        staticizer.generate(input, outputDir,
            project.staticizer.packageName);
```

```
      }
    }
  }
}
```

(from Gradle/Staticizer/staticizer/src/main/groovy/com/commonsware/android/staticizer/StaticizerTask.groovy)

This class has two fields. One, `outputDir`, is populated by `StaticizerPlugin` based on our build variant. The other, `inputDir`, is populated by:

- Getting the module's root directory via `getProjectDir()` on the `project` (where `project` is an inherited field pointing to the `Project`)
- Appending `src/main/staticizer` as being the directory where we expect to find the JSON files to convert into Java code

This does not take into account build variants, so we cannot have separate JSON in a `debug/` source set, for example. A fully production-grade version of this plugin would handle that.

The `@InputDirectory` and `@OutputDirectory` annotations tell Gradle the roles of these directories. In particular, Gradle uses this to determine whether our outputs are up to date with respect to our inputs, and therefore whether this task is needed or can be skipped.

The `execute()` method is annotated with `@TaskAction` and represents the entry point into the task. There, we:

- Confirm that we have a `packageName` in the `staticizer` extension object, though production-grade code would validate that this is a valid Java package name
- Create a Java `Staticizer` object that will do the actual code generation
- Iterate over all files in the `inputDir`, throw out those beginning with `.` (to filter out `.` and `..`, along with any leading-dot "hidden files"), and call `generate()` on the `Staticizer` for each

Production-grade code would confirm that the files are files and do something useful with subdirectories (e.g., recurse into them).

Between the `StaticizerPlugin` and `StaticizerTask`, we have implemented the Gradle plugin code. `Staticizer` itself knows nothing about Gradle. Its `generate()` method just knows that it is to take an input file, generate a Java output file, and use

a certain Java package while doing so. Hence, `Staticizer` could be used in other ways (e.g., command-line interface), tested separately, etc.

Generating Java Code

After all of that setup, the actual work of generating Java code is almost anti-climactic. Partly, that is due to the power of JavaPoet. Partly, that is due to the limited nature of the sample.

`Staticizer` is a plain Java class. Its `execute()` method is responsible for parsing the JSON and writing the corresponding Java code out to the designated location:

```
void generate(File input, File outputDir,
              String packageName) throws IOException {
  Type type=
    new TypeToken<LinkedHashMap<String,Object>>() {}.getType();
  LinkedHashMap<String, Object> data=
    new Gson().fromJson(new FileReader(input), type);
  String basename=removeExtension(input.getAbsolutePath());
  TypeSpec.Builder builder=TypeSpec.classBuilder(basename)
    .addModifiers(Modifier.PUBLIC, Modifier.FINAL);

  for (Map.Entry<String, Object> entry : data.entrySet()) {
    String fieldName=
      CaseFormat.LOWER_CAMEL
        .to(CaseFormat.UPPER_UNDERSCORE, entry.getKey());
    FieldSpec.Builder field;

    if (entry.getValue() instanceof Float) {
      field=FieldSpec.builder(TypeName.FLOAT, fieldName)
        .initializer("$L", entry.getValue());
    }
    else if (entry.getValue() instanceof Double) {
      field=FieldSpec.builder(TypeName.DOUBLE, fieldName)
        .initializer("$L", entry.getValue());
    }
    else if (entry.getValue() instanceof Integer) {
      field=FieldSpec.builder(TypeName.INT, fieldName)
        .initializer("$L", entry.getValue());
    }
    else if (entry.getValue() instanceof Long) {
      field=FieldSpec.builder(TypeName.LONG, fieldName)
        .initializer("$L", entry.getValue());
    }
    else if (entry.getValue() instanceof Boolean) {
      field=FieldSpec.builder(TypeName.BOOLEAN, fieldName)
```

```
        .initializer("$L", entry.getValue());
    }
    else {
      field=FieldSpec.builder(String.class, fieldName)
        .initializer("$S", entry.getValue().toString());
    }

    field.addModifiers(Modifier.PUBLIC, Modifier.STATIC, Modifier.FINAL)
      .build();

    builder.addField(field.build());
  }

  JavaFile.builder(packageName, builder.build())
    .build()
    .writeTo(outputDir);
}
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

Let's take this a piece at a time.

We have no idea what the JSON will look like, other than it should be a JSON object (versus being an array). The first few lines go through a typical Gson recipe to convert the JSON into a `Map` representing that object:

```
  Type type=
    new TypeToken<LinkedHashMap<String,Object>>() {}.getType();
  LinkedHashMap<String, Object> data=
    new Gson().fromJson(new FileReader(input), type);
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

The Java class name should match the JSON file name, just without the extension. The `removeExtension()` method is a simple algorithm for giving us the base name of the JSON file:

```
// inspired by http://stackoverflow.com/a/990492/115145

private static String removeExtension(String s) {
  String result;

  int sepIndex=
    s.lastIndexOf(System.getProperty("file.separator"));

  if (sepIndex==-1) {
    result=s;
```

```
    }
    else {
      result=s.substring(sepIndex+1);
    }

    int extIndex=result.lastIndexOf(".");

    if (extIndex!=-1) {
      result=result.substring(0, extIndex);
    }

    return(result);
  }
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

We are looking to create a Java class. For that, JavaPoet offers `TypeSpec.classBuilder()`, which gives us a `TypeSpec.Builder` set up to build a class with the supplied name. We call `addModifiers()` to indicate that the class should be `public` and `final` (meaning that the class cannot be subclassed):

```
    TypeSpec.Builder builder=TypeSpec.classBuilder(basename)
      .addModifiers(Modifier.PUBLIC, Modifier.FINAL);
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

We then iterate over all the entries in the `Map` loaded by Gson. For each, we want to define a field in our Java class. For that, we will need the field name, which should be the name of the attribute in the JSON object, but with the `camelCase` name replaced by `ITS_ALL_CAPS_EQUIVALENT`. Guava has a `CaseFormat` class that encapsulates this sort of conversion, so we indicate that we want to convert from "lower camel" formatting to "upper underscore" formatting:

```
      String fieldName=
        CaseFormat.LOWER_CAMEL
          .to(CaseFormat.UPPER_UNDERSCORE, entry.getKey());
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

We then need to set up a `FieldSpec.Builder` for the field that we want to add. Just as JavaPoet's `TypeSpec` generates a Java type (class, interface, etc.), `FieldSpec` generates a field within a type.

However, one of the parameters of the `FieldSpec.builder()` method is the data type of the resulting field. Gson has its own algorithm for determining what data

type to use for the values it parses. So, we use `instanceof` to determine the type of the attribute value, and we use that to determine how to set up the `FieldSpec.Builder`. For example, we use this code if the value is a `Double`:

```
      else if (entry.getValue() instanceof Double) {
        field=FieldSpec.builder(TypeName.DOUBLE, fieldName)
          .initializer("$L", entry.getValue());
      }
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

We tell `FieldSpec.builder()` to give us a `FieldSpec.Builder` set up to define a `double` field whose name is the `fieldName` that we got from the `CaseFormat` conversion of the attribute name. We specifically want that field to be initialized with the value itself, and for numeric or `boolean` fields, the JavaPoet syntax for that is to pass `$L` to `initializer()`, along with the value to be initialized.

If the value we got from Gson does not appear to be a number or a `boolean`, we treat it as a `String`:

```
      else {
        field=FieldSpec.builder(String.class, fieldName)
          .initializer("$S", entry.getValue().toString());
      }
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

Here, `$S` teaches JavaPoet to examine the `String` that we supply and add in any escape sequences that might be needed (e.g., for embedded quotation marks).

No matter what the type of the field is, we indicate that it should be `public static final` via an `addModifiers()` call, before adding the field to the class:

```
      field.addModifiers(Modifier.PUBLIC, Modifier.STATIC, Modifier.FINAL)
        .build();

      builder.addField(field.build());
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

After processing the entire JSON object, we create a `JavaFile.Builder`, teaching it the Java package name plus the type that goes into the file, and we write the resulting `JavaFile` out to `outputDir`:

```
    JavaFile.builder(packageName, builder.build())
      .build()
      .writeTo(outputDir);
```

(from Gradle/Staticizer/staticizer/src/main/java/com/commonsware/android/staticizer/Staticizer.java)

`JavaFile` will handle all the details of creating the appropriate set of subdirectories underneath `outputDir` based upon the Java package name.

Note that while developing this sort of logic, if you change your code-generation algorithm, you will need to clean your project as part of trying out the revised code. Gradle compares input and output to see if changes are needed, but it does not take the age of the plugin into account, and so it will not realize that your output would differ not because of changes to the input, but due to changes in the algorithm used to generate the output.

Using the Generated Code

The `app/` module in this project then uses the plugin, assuming that we have run `uploadArchives` in the plugin's module to publish the plugin to the local repository:

```
buildscript {
    repositories {
        jcenter()
        maven {
            url uri('../repository')
        }
    }
    dependencies {
        classpath 'com.commonsware.android:staticizer:0.0.1'
    }
}

apply plugin: 'com.android.application'
apply plugin: 'com.commonsware.android.staticizer'

staticizer {
    packageName 'com.commonsware.android.staticizer.demo'
}

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'
    defaultConfig {
        minSdkVersion 15
```

```
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
    }
}
```

(from Gradle/Staticizer/app/build.gradle)

We configure `packageName` on `staticizer` to be the same Java package that our activity source code happens to reside in, though that is not required.

We need to add the `staticizer/` directory to the `main/` source set and add in whatever JSON files we want, with appropriate filenames for the Java classes that we want to generate.

Then, our application code can refer to our custom-generated Java class (`TestData` in this case), the same way that it can refer to `R`, `BuildConfig`, or other Java classes generated by the standard Android build tools:

```
package com.commonsware.android.staticizer.demo;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    TextView tv=(TextView)findViewById(R.id.some_string);

    tv.setText(TestData.FOO);
    tv=(TextView)findViewById(R.id.version_code);
    tv.setText(Double.toString(TestData.VERSION_CODE));
  }
}
```

(from Gradle/Staticizer/app/src/main/java/com/commonsware/android/staticizer/demo/MainActivity.java)

Advanced Gradle for Android Tips

There are *lots* of things you can do given a full scripting language as the basis for your build system. This chapter represents a collection of tips for things that you can do that go beyond stock capabilities provided by the Android Gradle Plugin.

Prerequisites

Understanding this chapter requires that you have read the chapters that introduce Gradle and cover basic Gradle/Android integration, including the project structure.

Gradle, DRY

Ideally, your build scripts do not repeat themselves any more than is logically necessary. For example, a project and sub-projects probably should use the same version of the build tools, yet by default, we define them in each `build.gradle` file. This section outlines some ways to consolidate this sort of configuration.

It's build.gradle All The Way Down

If you have sub-projects, you can have `build.gradle` files at each level of your project hierarchy. Your top-level `build.gradle` file is also applied to the sub-projects when they are built.

In particular, you can "pass data" from the top-level `build.gradle` file to sub-projects by configuring the `ext` object via a closure. In the top-level `build.gradle` file, you would put common values to be used:

```
ext {
   compileSdkVersion=26
}
```

(note the use of the = sign here)

Sub-projects can then reference `rootProject.ext` to retrieve those values:

```
android {
    compileSdkVersion rootProject.ext.compileSdkVersion
}
```

By this means, you can ensure that whatever needs to be synchronized at build time is synchronized, by defining it once.

Another way that a top-level `build.gradle` file can configure subprojects is via the `subprojects` closure. This contains bits of configuration that will be applied to each of the subprojects as a part of their builds.

Note that `subprojects` applies to *all* sub-projects (a.k.a., modules), which limits its utility. For example, a top-level project with one sub-project for an app and another sub-project for a library used by that app cannot readily use `subprojects`. That is because the library sub-project needs to configure the `com.android.library` plugin, while the application sub-project needs to configure the `com.android.application` plugin. The `subprojects` closure is only good for common configuration to apply to all sub-projects regardless of project type.

gradle.properties

Another approach would be to add a `gradle.properties` file to your project root directory. Those properties are automatically read in and would be available up and down your project hierarchy.

Per-developer properties can go in a `gradle.properties` file in the user's Gradle home directory (e.g., `~/.gradle` on Linux), where they will not be accidentally checked into version control.

So, to achieve the synchronized `compileSdkVersion` value, you could have a `gradle.properties` file with:

```
COMPILE_SDK_VERSION=26
```

Then, your projects' `build.gradle` files could use:

```
android {
    compileSdkVersion COMPILE_SDK_VERSION
}
```

Custom Properties Files

You are also welcome to use your own custom properties files. For example, perhaps you want to use `gradle.properties` for properties that you are willing to put in version control (e.g., `BUILD_TOOLS_VERSION`), but you would *also* like to use a properties file to keep your code-signing details outside of your `build.gradle` file and out of version control.

Loading in custom properties files is slightly clunky, as it does not appear to be built into Gradle itself. However, you can take advantage of the fact that Gradle is backed by Groovy and use some ordinary Groovy code to load the properties.

```
def keystorePropertiesFile = file('keystore.properties')
def keystoreProperties = new Properties()
keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
```

These three lines create a `java.util.Properties` object from a `keystore.properties` file located in the module's directory. Individual properties can be read using <> syntax, the same as Groovy uses for a `Map`:

```
signingConfigs {
  release {
    keyAlias keystoreProperties['keyAlias']
    keyPassword keystoreProperties['keyPassword']
    storeFile file(keystoreProperties['storeFile'])
    storePassword keystoreProperties['storePassword']
  }
}
```

This fills in the signing configuration from the `keystore.properties` values, assuming that the `keystore.properties` file itself has the appropriate properties:

```
storePassword=81016168
keyPassword=9f353470
keyAlias=foo
storeFile=release.jks
```

Now, your signing information is not in `build.gradle`, and you can keep a fake `keystore.properties` in version control (where the properties have invalid values), enough to allow Gradle to process the `build.gradle` file but not allow for app signing. The machine designated for doing official builds would have the real `keystore.properties` file and associated keystore.

Environment Variables

Any environment variables with a prefix of `ORG_GRADLE_PROJECT_` will show up as global variables in your Gradle script. So, for example, you can access an environment variable named `ORG_GRADLE_PROJECT_foo` by accessing a `foo` variable in `build.gradle`.

If you would prefer to use environment variables without that prefix, you can call `System.getenv()`, passing in the name of the environment variable, to retrieve its value.

Note, however, that you may or may not have access to the environment variables that you think you should. Android Studio, for example, does not expose environment variables to Gradle for its builds, and so an environment variable that you can access perfectly well from the command line may not be available in the same `build.gradle` script when run from Android Studio.

Automating APK Version Information

Once the Android Gradle Plugin started catching on, one of the first things many developers raced to do was automate the `android:versionCode` and `android:versionName` properties from the manifest. Since those can be defined in a Gradle file (overriding values from any `AndroidManifest.xml` files), and since Gradle is backed by Groovy, it is possible to programmatically assign values to those properties.

This section outlines a few approaches to that problem.

Auto-Incrementing the versionCode

Since the `android:versionCode` is a monotonically increasing integer, one approach for automating it is to simply increment it on each build. While this may seem wasteful, two billion builds is a *lot* of builds, so a solo developer is unlikely to run out. Synchronizing such `versionCode` values across a team will get a bit more

complex, but for an individual case (developer, build server, etc.), it is eminently doable using Groovy.

The Gradle/Versioning sample project uses a `version.properties` file as the backing store for the version information:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 26
    buildToolsVersion '26.0.2'

    def versionPropsFile = file('version.properties')
    def Properties versionProps = new Properties()

    if (versionPropsFile.canRead()) {
        versionProps.load(new FileInputStream(versionPropsFile))
    }
    else {
        versionProps['VERSION_CODE']='0'
    }

    def code = versionProps['VERSION_CODE'].toInteger() + 1

    versionProps['VERSION_CODE']=code.toString()
    versionProps.store(versionPropsFile.newWriter(), null)

    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 26
        versionCode code
        versionName "1.1"
    }
}
```

(from Gradle/Versioning/app/build.gradle)

First, we try to open a `version.properties` file. If we find it, we read it in. Otherwise, we initialize our `java.util.Properties` object with a `VERSION_CODE` of `0`, simulating a starting point.

The script then increments the old value by 1 to get the new code to use. The revised code is then written back to the properties file before it is applied in the `defaultConfig` closure.

The result is that our `versionCode` is automatically incremented, and you can see the value change in the `version.properties` file that is generated into the `app/` module directory:

```
#Sat Nov 04 12:44:54 EDT 2017
VERSION_CODE=3
```

(from Gradle/Versioning/app/version.properties)

Adding to BuildConfig

The Android development tools have been code-generating the `BuildConfig` class for some time now. Historically, the sole element of that class was the `DEBUG` flag, which is `true` for a debug build and `false` otherwise. This is useful for doing runtime changes based upon build type, such as only configuring `StrictMode` in debug builds.

Nowadays, the Android Gradle Plugin also defines things like:

- `BUILD_TYPE`, which is the build type used to build this APK.
- `FLAVOR`, which is the product flavor used to build this APK.
- `APPLICATION_ID`, which is the name that serves as the application ID (i.e., it includes build type suffixes and product flavor overrides). This is useful for cases where you cannot just call `getPackageName()` on a `Context` because you do not have a handy `Context`.
- `VERSION_CODE`, which is the version code derived from your manifest in conjunction with any overrides coming from your `build.gradle` file.
- `VERSION_NAME`, which is the version name derived from your manifest in conjunction with any overrides coming from your `build.gradle` file.

However, you can add your own data members to `BuildConfig`, by including a `buildConfigField` statement in the `defaultConfig` closure of your `android` closure:

```
android {
  defaultConfig {
    buildConfigField "int", "FOO", '5'
  }
}
```

You can use this to embed any sort of information you want into `BuildConfig`, so long as it is knowable at compile time.

Moreover, you can also have `buildConfigField` statements in build types. This would be useful if you have custom build types, beyond just debug and release, and you need runtime configuration for those. For example, you could put server URLs in `buildConfigField`, so your debug server is different from your integration test server, which in turn is different than your production server.

For example, in the chapter on SSL, we will see a sample app for demonstrating network security configuration, a way that apps can restrict their Internet access (e.g., require SSL for everything) and support custom SSL certificates (e.g., self-signed certificates for an internal test server). The `Internet/CA` sample project that we will see there has a long list of product flavors, and each of them defines a particular server URL to test against:

```
apply plugin: 'com.android.application'

def WARES='"https://wares.commonsware.com/excerpt-7p0.pdf"'
def SELFSIGNED='"https://scrap.commonsware.com:3001/excerpt-7p0.pdf"'

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 17
        targetSdkVersion 27
    }

    flavorDimensions "default"

    productFlavors {
        comodo {
            dimension "default"
            resValue "string", "app_name", "CA Validation Demo"
            applicationId "com.commonsware.android.downloader.ca.comodo"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_comodo']
            buildConfigField "String", "URL", WARES
        }
        verisign {
            dimension "default"
            resValue "string", "app_name", "Invalid CA Validation Demo"
            applicationId "com.commonsware.android.downloader.ca.verisign"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_verisign']
            buildConfigField "String", "URL", WARES
        }
```

```
        system {
            dimension "default"
            resValue "string", "app_name", "System CA Validation Demo"
            applicationId "com.commonsware.android.downloader.ca.system"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_verisign_system']
            buildConfigField "String", "URL", WARES
        }
        pin {
            dimension "default"
            resValue "string", "app_name", "Cert Pin Demo"
            applicationId "com.commonsware.android.downloader.ca.pin"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_pin']
            buildConfigField "String", "URL", WARES
        }
        invalidPin {
            dimension "default"
            resValue "string", "app_name", "Cert Pin Demo"
            applicationId "com.commonsware.android.downloader.ca.invalidpin"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_invalid_pin']
            buildConfigField "String", "URL", WARES
        }
        selfSigned {
            dimension "default"
            resValue "string", "app_name", "Self-Signed Demo"
            applicationId "com.commonsware.android.downloader.ca.ss"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_selfsigned']
            buildConfigField "String", "URL", SELFSIGNED
        }
        override {
            dimension "default"
            resValue "string", "app_name", "Debug Override Demo"
            applicationId "com.commonsware.android.downloader.ca.debug"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_override']
            buildConfigField "String", "URL", SELFSIGNED
        }
    }
}

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}
```

```
dependencies {
    implementation 'com.android.support:support-v13:27.0.2'
    implementation 'com.commonsware.cwac:provider:0.5.3'
    implementation 'com.commonsware.cwac:netsecurity:0.4.4'
}
```

(from Internet/CA/app/build.gradle)

In this case, to reduce repetition, the two possible values are defined as globals and are poured into `BuildConfig.URL` via the `buildConfigField` statements.

Testing with JUnit4

Presumably, you will want to test your code, beyond just playing around with it yourself by hand. Android offers several means of testing your app, covered in this next series of chapters.

The first Android SDK testing solution we will examine is the JUnit test framework. This is a standard Java unit testing framework. Originally, Android itself "baked in" a copy of JUnit3. This has since been deprecated, and modern Android testing is done with a separate copy of JUnit4, in the form of a `AndroidJUnitRunner` class.

In this chapter, we will review how to apply the `AndroidJUnitRunner` to run JUnit4 tests for our Android apps.

Prerequisites

Understanding this chapter requires that you have read the core chapters of the book.

This chapter also assumes you have some familiarity with JUnit, though you certainly do not need to be an expert. You can learn more about JUnit at the JUnit site.

Instrumentation Tests and Unit Tests

There are two places in Android app development where we use JUnit4: instrumentation tests and unit tests. Both serve the same objective: confirm that our code runs as expected. What differs in where the code lives (`androidTest` versus

`test` source sets) and where the code runs (inside of Android or on your development machine directly).

The following sections outline the differences between the two, though there is a separate chapter dedicated to unit testing, with the bulk of this chapter focused on instrumentation testing.

Where Your Test Code Lives

One common problem with testing is determining where the test code should reside, relative to the production code being tested. Ideally, these are not intermingled, as that would increase the odds that you might accidentally ship the testing code as part of your production app — at best, this increases your APK size; at worst, it could open up security flaws.

With Gradle-based projects, including those created for Android Studio, we have a dedicated source set for our instrumentation tests, named `androidTest`, where the code for those tests would reside.

As with any source set, `androidTest` can have Java code, resources, etc. It does not need an `AndroidManifest.xml` file, though, as that will be auto-generated.

Unit tests, by contrast, will go in a `test` source set.

Where Your Test Code Runs

Ordinarily, each code base (e.g., project) is packaged in its own APK and is executed in its own process.

In the case of instrumentation tests, your test code and your production code are combined into a single process in a single copy of the virtual machine.

This will allow your JUnit test methods to access objects from your production code, such as your activities and their widgets.

However, this does limit instrumentation testing to be run from a developer's computer. You cannot package JUnit tests to be initiated from the device itself, except perhaps on rooted devices.

Unit tests, on the other hand, bypass Android and run straight on your development machine. As a result, they cannot use much of the Android SDK, and so these tests

are limited in terms of what they can test. However, they will run much more quickly, and so it may be worthwhile to set up a subset of your tests as unit tests.

Writing JUnit4 Test Cases

As noted in the intro to the chapter, modern Android testing — both instrumentation testing and unit testing — is done through JUnit4.

This book does not attempt to cover all aspects of JUnit4. For that, you are encouraged to read the JUnit documentation or other books on Java testing. This chapter will cover some of the basics of using JUnit4 tests, plus some of the issues with using JUnit4 tests in Android.

The Class

In JUnit terminology, "test case" is a Java class that represents a set of tests to run.

Any Java class can serve as a test case, so long as it has a zero-argument public constructor and is known to the test runner that it contains tests to be run. In the case of JUnit4 on Android, that comes via a `@RunWith(AndroidJUnit4.class)` annotation on the class, to signal that this class contains tests:

```
@RunWith(AndroidJUnit4.class)
public class ICanHazTests {
  // test code goes here
}
```

The Test Methods

In JUnit, a "test method" is a method, in a test case, that tests something in some production code base.

In JUnit4, a test method is any public method that is annotated with the `@Test` annotation:

```
@RunWith(AndroidJUnit4.class)
public class ICanHazTests {
  @Test
  public void kThxBye() {
    // do some testing
  }
}
```

A test method can then execute code to see if it works, and "assert" some conditions ("the response from the `foo()` method should be `1`"). JUnit4 supplies an `Assert` class with static assertion methods that we can employ:

```
@RunWith(AndroidJUnit4.class)
public class SillyTest {
  @Test
  public void thisIsReallySilly() {
    Assert.assertEquals("bit got flipped by cosmic rays", 1, 1);
  }
}
```

`assertEquals()` takes either two parameters of the same type for comparison (e.g., two `int` values), or three parameters, where the first is a custom assertion failure message.

There are countless other methods on `Assert` (e.g., `assertNotNull()`) for testing objects, collections, etc.

Setup and Teardown

A JUnit test case can also have methods that represent "setup" and "teardown" work. A "setup" method is one that executes before test methods and helps establish a common environment to be used by all of the test methods. A "teardown" method is one that is run after test methods and is used to clean up things created by the "setup" method. The objective is to ensure that each test method has a consistent and expected environment (e.g., contents of databases).

In JUnit4, you can annotate methods with `@Before` and `@After` for per-test-method setup and teardown work. The `@Before` method will be invoked before each test method is called; the `@After` method will be invoked after each test method is called.

JUnit4 also offers static `@BeforeClass` and `@AfterClass` methods, which are invoked once for the entire test case, designed for setting up immutable starter data for test methods and avoiding the overhead of doing that work on each test method invocation.

The `Testing/JUnit4` sample project illustrates the basics of setting up JUnit4 instrumentation tests.

We start off with a test case that is, well, silly:

```
package com.commonsware.android.abf.test;

import android.support.test.runner.AndroidJUnit4;
import junit.framework.Assert;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class SillyTest {
  @BeforeClass
  static public void doThisFirstOnlyOnce() {
    // do initialization here, run once for all SillyTest tests
  }

  @Before
  public void doThisFirst() {
    // do initialization here, run on every test method
  }

  @After
  public void doThisLast() {
    // do termination here, run on every test method
  }

  @AfterClass
  static public void doThisLastOnlyOnce() {
    // do termination here, run once for all SillyTest tests
  }

  @Test
  public void thisIsReallySilly() {
    Assert.assertEquals("bit got flipped by cosmic rays", 1, 1);
  }
}
```

(from Testing/JUnit4/app/src/androidTest/java/com/commonsware/android/abf/test/SillyTest.java)

All we have is a single test method — `thisIsReallySilly()` — that validates that 1 really does equal 1. Fortunately, this test usually succeeds. Our `SillyTest` also implements `@Before`, `@After`, `@BeforeClass`, and `@AfterClass` methods for illustration purposes, as there is little preparation needed for our rigorous and demanding test method.

Testing Activities

JUnit4 offers "test rules", which are packaged bits of reusable code for testing certain scenarios. For example, the Android `rules` artifact has an `ActivityTestRule` to help you test your activities.

For example, `DemoActivityRuleTest` tests an activity from the main app, where the activity has a `ListView` with 25 Latin words in it:

```
package com.commonsware.android.abf.test;

import android.support.test.rule.ActivityTestRule;
import android.support.test.runner.AndroidJUnit4;
import android.widget.ListView;
import com.commonsware.android.abf.ActionBarFragmentActivity;
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class DemoActivityRuleTest {
  private ListView list=null;
  @Rule public final ActivityTestRule<ActionBarFragmentActivity> main
      =new ActivityTestRule(ActionBarFragmentActivity.class, true);

  @Before
  public void init() {
    list=(ListView)main.getActivity().findViewById(android.R.id.list);
  }

  @Test
  public void listCount() {
    Assert.assertEquals(25, list.getAdapter().getCount());
  }
}
```

(from Testing/JUnit4/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

The `@Rule` annotation tells JUnit4 that this data member represents a JUnit4 rule that should be applied to the tests in this test case. `ActivityRuleTest` takes over the work of creating and destroying an instance of our `ActionBarFragmentActivity` as part of standard `@Before` and `@After` processing. The `true` in the `ActivityTestRule` constructor simply indicates that we want the activity to start off in touch mode.

We then use a `@Before` method to retrieve the `ListView` itself. We retrieve the activity created by the rule by calling `getActivity()` on the rule itself (here called `main`). Anything that is `public` on our activity — including most of the methods that we inherit from `Activity` — is usable here, such as `findViewById()`.

Our test method — `listCount()` — just confirms that our `ListAdapter` has 25 items in it.

Testing Context-Dependent Code

Sometimes, you do not need an activity, just some `Context`, for testing code that takes one as input (e.g., file I/O, database I/O, resources, assets). In those cases, you can just create a plain JUnit4 test case, but use the `InstrumentationRegistry` to get at a suitable `Context` for your test methods.

Specifically, wherever you need a `Context` tied to your app that you are testing, call `InstrumentationRegistry.getTargetContext()`.

The `InstrumentationRegistry` also has `getInstrumentation()` (which returns the `Instrumentation` object that we are using for testing) and `getContext()` (which returns the `Context` for our test code's package).

`DemoContextTest` demonstrates this:

```
package com.commonsware.android.abf.test;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import android.test.AndroidTestCase;
import android.test.UiThreadTest;
import android.view.LayoutInflater;
import android.view.View;
import com.commonsware.android.abf.R;
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class DemoContextTest {
  private View field=null;
  private View root=null;

  @Before
```

```
  public void init() {
    InstrumentationRegistry.getInstrumentation().runOnMainSync(new Runnable() {
      @Override
      public void run() {
        LayoutInflater inflater=LayoutInflater
            .from(InstrumentationRegistry.getTargetContext());

        root=inflater.inflate(R.layout.add, null);
      }
    });

    root.measure(800, 480);
    root.layout(0, 0, 800, 480);

    field=root.findViewById(R.id.title);
  }

  @Test
  public void exists() {
    Assert.assertNotNull(field);
  }

  @Test
  public void position() {
    Assert.assertEquals(0, field.getTop());
    Assert.assertEquals(0, field.getLeft());
  }
}
```

(from Testing/JUnit4/app/src/androidTest/java/com/commonsware/android/abf/test/DemoContextTest.java)

Here, we manually inflate the contents of the `res/layout/add.xml` resource, and lay them out as if they were really in an activity, via calls to `measure()` and `layout()` to simulate a WVGA800 display. At that point, we can start testing the widgets inside of that layout, from simple assertions to confirm that they exist, to testing their size and position.

Note that the act of inflating the layout is performed inside a `Runnable`, which itself is passed to `runOnMainSync()` on an `Instrumentation`. `runOnMainSync()` says "run this code on the main application thread, then block the current thread until that code has completed". On some versions of Android, layout inflation needs to happen on the main application thread, and therefore the test is more reliable if we do that inflation via `runOnMainSync()`. Test methods themselves run on a background thread, not the main application thread.

Configuring Gradle

Beyond having test code, we also need to provide some configuration information to Gradle to allow us to run these tests, eventually.

The Test Dependency

First, you need to add a test dependency — a dependency that will only be used as part of instrumentation testing. That can be accomplished via an `androidTestImplementation` statement in the `dependencies` closure, instead of a `implementation` statement, to limit the scope of the dependency to the case where the `androidTest` source set is in use.

Specifically, we need the `com.android.support.test:rules` artifact from the Android Support Repository:

```
dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    androidTestImplementation 'com.android.support.test:rules:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

(from Testing/JUnit4/app/build.gradle)

As of the time of this writing, the latest version of this artifact is `0.5`.

However, the `rules` artifact also depends upon some other artifacts available in public repositories, like Maven Central or Bintray's JCenter. If you have one of those set up already — and a typical Android Studio project will via the top-level `build.gradle` file — then Gradle will be able to search those repositories for the dependencies.

The Test Runner

A "test runner", in JUnit terms, is a piece of code that knows how to plug into JUnit, execute tests, and collect any exceptions or assertion failures that result from those tests.

JUnit4 uses a test runner named `AndroidJUnitRunner`, which we gain access to through the aforementioned test dependency.

In the `defaultConfig` closure, we can teach Gradle to use that test runner, via the `testInstrumentationRunner` value:

```
android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        testApplicationId "com.commonsware.android.gradle.hello.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

(from Testing/JUnit4/app/build.gradle)

The Test Application ID

That `defaultConfig` closure also specifies a `testApplicationId`. This is a replacement value for our `applicationId` when we are running our tests. This allows our test build to run without disturbing any other builds (e.g., standard `debug` builds) on our test device or emulator.

The convention is to have the `testApplicationId` be your regular `applicationId` (or `package` from your `<manifest>`) with `.test` on the end, but that is merely a convention.

Running Your Instrumentation Tests

Writing tests is nice. Running tests is nicer. Hence, it would be useful if we could run our JUnit4 tests.

Android Studio Ad-Hoc Test Runs

You have few options for quickly running one of your instrumentation tests.

In the Android Studio editor for your test case, each test method will have a green “run” icon in the gutter. Clicking that will run that test method on your chosen device or emulator:

```
    @Test
    public void listCount() {
      Assert.assertEquals(25, list.getAdapter().getCount());
    }
}
```

Figure 347: Android Studio Per-Method Test Option

You can also right-click over a test case class, or a Java package containing test cases, and choose to run the tests cases from the context menu:

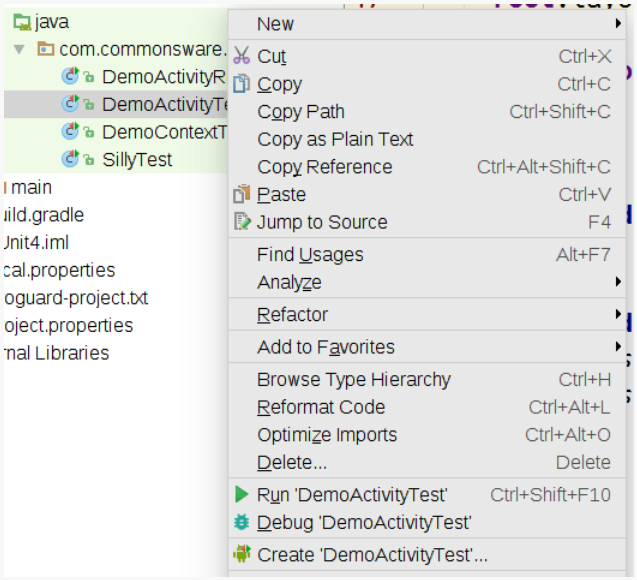


Figure 348: Android Studio Per-Class or Per-Package Test Option

Android Studio Run Configuration

You will see your ad-hoc runs appear in the drop-down list to the left of the run button in the Android Studio toolbar. That drop-down list represents your "run configurations", and you can set one of those up yourself directly if you wish.

To do that, choose Run > "Edit Configurations" from the main menu. That will bring up a dialog showing your current run configurations:

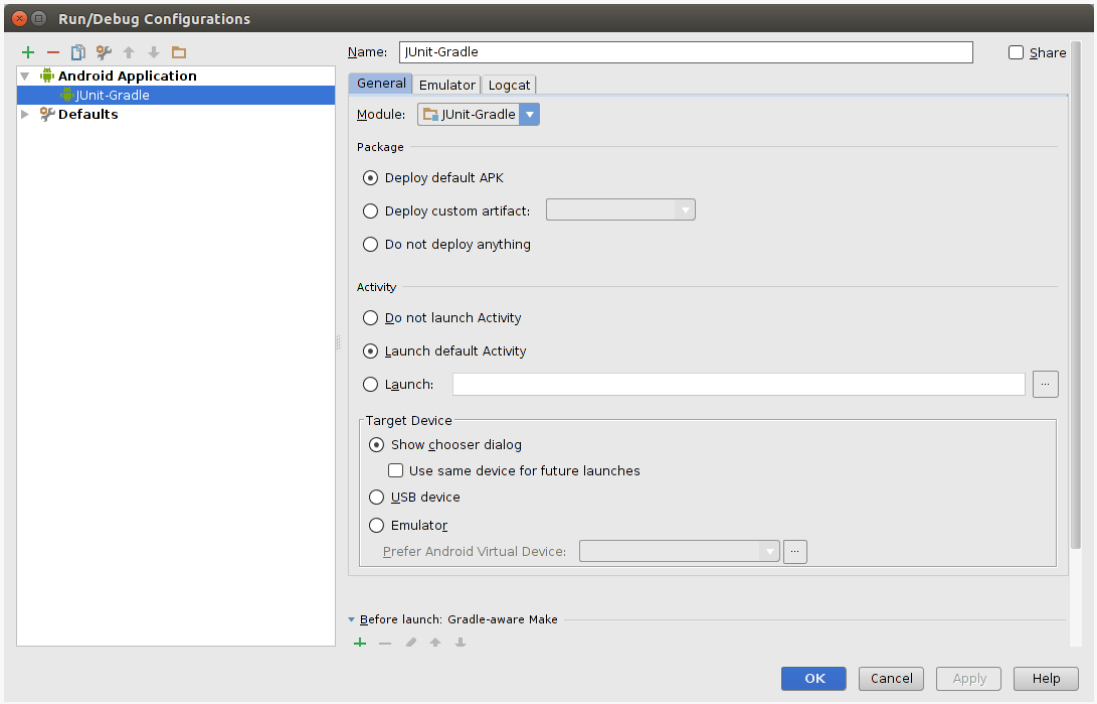


Figure 349: Android Studio Run Configurations Dialog

Towards the upper-left corner of the dialog, you will see a green plus sign. Tapping that will drop down a list of configuration types to choose from:

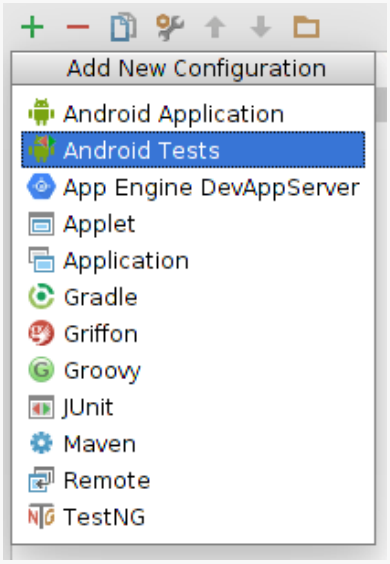


Figure 350: Android Studio, Adding a New Run Configuration

Choose “Android Instrumented Tests”, and a new empty configuration will be set up for you. You can name it whatever you want via the “Name” field (e.g., “Instr Tests”). Choose your project’s module that you wish to test in the “Module” drop-down (e.g., app). You can also choose the scope of the testing (e.g., “All in Module”), where to run the tests (e.g., “Show chooser dialog”), plus other settings.

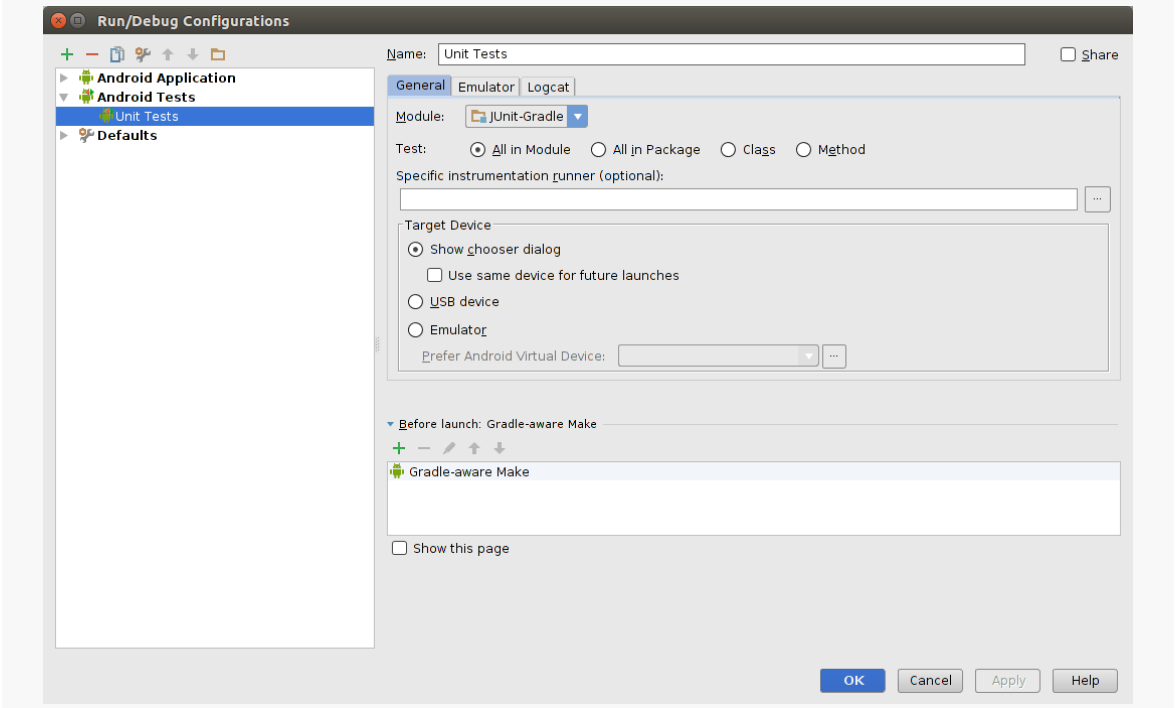


Figure 351: Android Studio, Showing New “Tests” Run Configuration

At that point, you can choose your run configuration from the drop-down to the left of the “play” button in the toolbar:

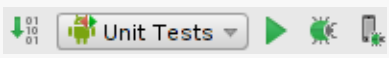


Figure 352: Android Studio Toolbar, Showing “Tests” Run Configuration

Note that the context menu for a class or package containing test cases has a “Create ...” option for creating a test run configuration specific for that class or package.

Examining the Test Results

Regardless of how you run the tests, the output will be shown in the Run view, normally docked in the bottom of your IDE window:

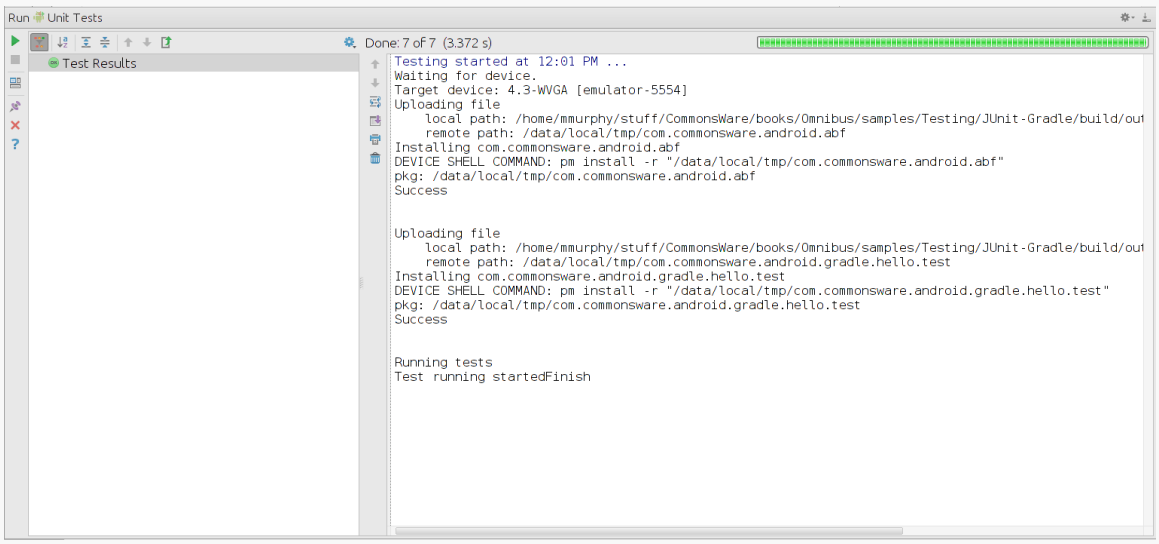


Figure 353: Android Studio, Showing Run Unit Tests Results

If a test fails an assertion or crashes, the test results will show the test case and test method that failed, along with the associated stack trace:

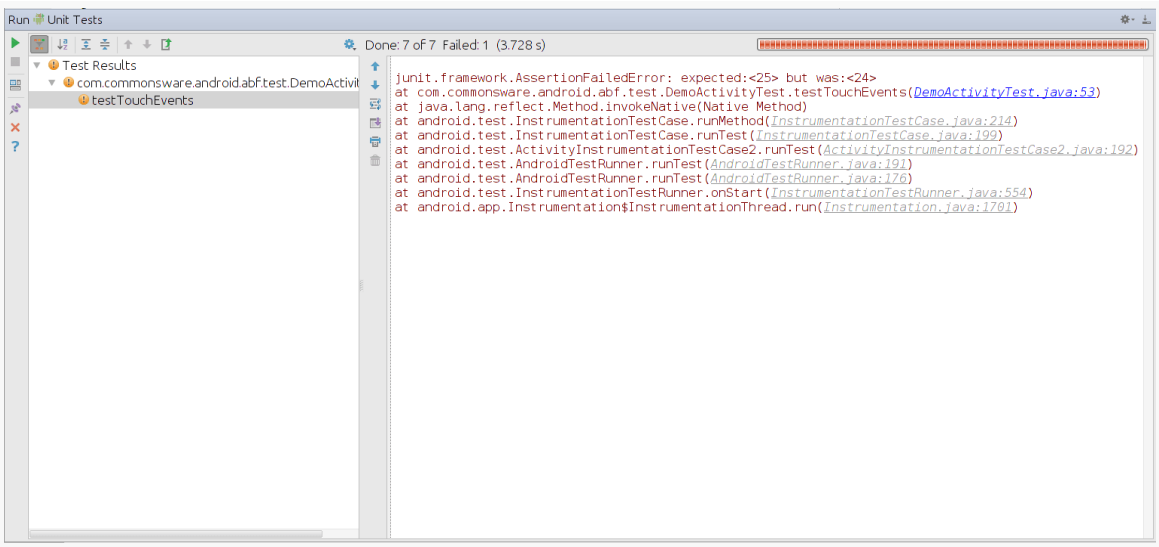


Figure 354: Android Studio, Showing Run Unit Tests Results With a Failure

Gradle for Android

The primary Gradle task that you will use related to testing is `connectedCheck`. This task will build the main app, then, build the test app (using a generated manifest to go along with the code from your `androidTest` source set).

At that point, the task will iterate over *all* compatible connected devices and running emulator instances. For each such Android environment, the task will install both apps, run the tests, and uninstall both apps.

Raw test results, in XML format, will be written to `build/outputs/androidTest-results/connected`. These will primarily be of interest to toolsmiths, such as those adding support for Android Gradle-based builds to continuous integration (CI) servers.

For others, the HTML reports will be of greater use. These will be written to `build/outputs/reports/androidTests/connected`, with an `index.html` file serving as your entry point. These will show the results of all of your tests, with hyperlinked pages to be able to "drill down" into the details, such as to investigate failed tests.

Testing Android Library Projects

The above procedures are aimed at testing Android application projects. If you are creating an Android library project, you can also use JUnit for testing.

A Gradle-built Android library project can have an `androidTest` source set, just like a regular app. And, a Gradle-built Android library project can be tested via the `connectedCheck` task. However, that task will create and install a single APK, consisting of the code from the `androidTest` source set combined with the library project's own code.

From the standpoint of what you do as a developer, though, it works just like testing an app: add your test cases to the `androidTest` source set and use `connectedCheck` to run the tests.

Testing and Runtime Permissions

Android 6.0 added the concept of runtime permissions, where permissions with a `protectionLevel` of `dangerous` are not granted automatically, but instead need to be

requested at runtime. Those requests result in a system-supplied dialog appearing, one that the user is supposed to tap on to grant or deny your request.

Our instrumentation tests cannot interact with this dialog. Moreover, our test code is its own separate app, and the first time it is installed, it will not have the runtime permissions yet. Hence, if we try testing code that needs those permissions, our tests fail.

There are two major approaches for dealing with this. If you wish to test the actual UI flow of your runtime permission request logic, you will need to use UiAutomator to simulate user clicks on the permission dialog. If, however, your instrumentation tests are focused on logic beneath the UI layer, you can grant the permissions yourself in your test code, bypassing any need for the dialog.

This is merely a matter of having the following code snippet in your test classes:

```
if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
  InstrumentationRegistry
    .getInstrumentation()
    .getUiAutomation()
    .executeShellCommand(String.format("pm grant %s
android.permission.READ_EXTERNAL_STORAGE",
      target.getPackageName()));
}
```

Here, you would replace `READ_EXTERNAL_STORAGE` with whatever runtime permission you need, and execute multiple commands if you need to grant multiple permissions.

If you put this in `@BeforeClass`-annotated methods on your test case classes that need runtime permissions, you will minimize the amount of time spent executing this command, while ensuring that the runtime permissions are granted by the time your test methods exercise code that needs those permissions.

The Android Test Orchestrator

By default, when we run our instrumentation tests, Android forks a single process and runs all the tests in that process. This has two problems:

1. Results from previous tests might affect later tests. For example, perhaps a later test would have caught a bug, but only if some `static` field were not initialized... and it was initialized in some earlier test.

2. If we crash with an unhandled exception, particularly in a background thread, our test suite may terminate.

Quietly released in 2017, the Android Test Orchestrator addresses this by running each test method in its own separate process. This clears up these two problems, though it adds quite a bit of overhead. In particular, anything that you do in a custom `Application` subclass, such as in its `onCreate()` method, now happens for *every test method*, rather than just once for each run of your tests.

For Android Studio and Gradle-based tests, setting up the Android Test Orchestrator is fairly straightforward:

- Make sure that you are using `1.0.1` or higher of the `com.android.support.test:runner` library
- Add `androidTestUtil 'com.android.support.test:orchestrator:1.0.1'` to your `dependencies`, where the version number should match that of the `com.android.support.test:runner` dependency:

```
dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestUtil 'com.android.support.test:orchestrator:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
}
```

(from Testing/Orchestrator/app/build.gradle)

- Add a `testOptions` closure to the `android` closure of your module's `build.gradle` file, requesting the `execution` of `ANDROID_TEST_ORCHESTRATOR`:

```
android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        testApplicationId "com.commonsware.android.gradle.hello.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }

    testOptions {
        execution 'ANDROID_TEST_ORCHESTRATOR'
    }
}
```

(from Testing/Orchestrator/app/build.gradle)

And that's it. There are no Java code changes required — you write your tests normally.

The Testing/Orchestrator sample project is cloned from the `JUnit4` sample, with the aforementioned changes, plus a custom `Application` class:

```
package com.commonsware.android.abf;

import android.app.Application;
import android.util.Log;

public class ScrapApp extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    Log.e("Frosty the Snowman", "Happy Birthday!");
  }
}
```

(from Testing/Orchestrator/app/src/main/java/com/commonsware/android/abf/ScrapApp.java)

Without the Android Test Orchestrator, the "Happy Birthday!" message would appear once in LogCat for the entire test suite. With the Android Test Orchestrator, the message appears once for each test method... and, inexplicably, an eighth time as well:

```
12-18 07:38:11.356 11131-11131/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:12.438 11159-11159/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:14.023 11213-11213/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:15.358 11260-11260/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:18.454 11316-11316/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:19.809 11366-11366/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:20.711 11392-11392/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
12-18 07:38:21.635 11418-11418/com.commonsware.android.abf E/Frosty the Snowman:
Happy Birthday!
```

The downside comes in the speed of the tests. Running this short suite under the Android Test Orchestrator is noticeably slower than running the suite without the

Orchestrator. A test suite measuring in the thousands of test methods will result in a fair bit of additional overhead, forking thousands of processes instead of just one.

Testing with Espresso

Basic JUnit4 instrumentation tests are fine for testing non-UI logic. They even work acceptably for some basic UI testing. The more complex your UI testing gets, though, the more likely it is that you will find plain JUnit4 instrumentation tests to be limiting and tedious.

In particular, running tests across activities can be tricky with ordinary JUnit4. `ActivityTestRule` is designed for testing a single activity in isolation, and crafting your own rule that transcends a single activity may be difficult.

Espresso is designed to simplify otherwise-complex UI testing scenarios, such as:

- Testing across activities, such as confirming that tapping a `ListView` row in one activity correctly launches a detail activity associated with the model object for that row
- Testing over time, such as waiting for a list to be populated from a database before actually testing it

In this chapter, we will explore how to set up basic Espresso tests and how to employ them as part of your overall testing implementation.

Prerequisites

This chapter assumes that you have read the chapter on JUnit4.

Adding a Shot of Espresso

The Testing/Espresso sample project is the home of several test cases that employ Espresso, so we can see how it works in practice.

The app/ module's build.gradle file is fairly conventional, reminiscent of our JUnit4 equivalent, except that we have several dependencies:

```
apply plugin: 'com.android.application'

dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    androidTestImplementation 'com.android.support:support-annotations:27.0.2'
    androidTestImplementation 'com.android.support:recyclerview-v7:27.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-contrib:3.0.1'
    implementation 'com.android.support:recyclerview-v7:27.0.2'
}

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        applicationId "com.commonsware.android.espresso"
        testApplicationId "com.commonsware.android.espresso.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        testInstrumentationRunnerArguments disableAnalytics: 'true'
    }

    packagingOptions {
        exclude 'LICENSE.txt'
    }
}
```

(from Testing/Espresso/app/build.gradle)

The sample app has activities for a ListView and a RecyclerView, and so we implementation in the recyclerview-v7 dependency for that reason. However, beyond that, there are four androidTest dependencies, to pull in things needed for instrumentation testing.

The big one is espresso-core, which contains the bulk of the Espresso test engine. Through Gradle's transitive dependencies, pulling in espresso-core also pulls in other key testing artifacts, such as:

- com.android.support.test:runner
- com.android.support.test:rules

- `com.android.support:support-annotations`
- `junit:junit` (indirectly, via `com.android.support.test:runner`)

Hence, just by asking for `espresso-core`, we pull in everything that we need not only for basic Espresso testing but also for general JUnit4-style instrumentation testing.

However, the `support-annotations` requested by `espresso-core` is an older version than the one that we are using in our code under test. So, we specifically pull in the newer `support-annotations`, allowing Gradle's dependency resolution to handle the version discrepancy.

The `espresso-contrib` and `recyclerview-v7` `androidTest` dependencies are for testing `RecyclerView`. Those artifacts will be discussed later in this chapter.

The test cases themselves still reside in the `androidTest/` source set and still use `@RunWith(AndroidJUnit4.class)`. Those aspects of instrumentation testing have not changed, just because we are using Espresso. And we are able to write classic Espresso-free instrumentation tests as well — we are not forced to use Espresso for everything, just because Espresso is part of our environment. So, for example, `SillyTest` in this sample project is the same as before:

```
package com.commonsware.android.abf.test;

import android.support.test.runner.AndroidJUnit4;
import junit.framework.Assert;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class SillyTest {
  @BeforeClass
  static public void doThisFirstOnlyOnce() {
    // do initialization here, run once for all SillyTest tests
  }

  @Before
  public void doThisFirst() {
    // do initialization here, run on every test method
  }
```

```
  @After
  public void doThisLast() {
    // do termination here, run on every test method
  }

  @AfterClass
  static public void doThisLastOnlyOnce() {
    // do termination here, run once for all SillyTest tests
  }

  @Test
  public void thisIsReallySilly() {
    Assert.assertEquals("bit got flipped by cosmic rays", 1, 1);
  }
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/SillyTest.java)

Writing Tests in Espresso

Writing Espresso tests is often described as having three main steps:

1. Find the widgets you want to examine or manipulate
2. Perform actions on those widgets where needed (and where possible)
3. Check to see if widgets have a certain state

Finding Widgets via Hamcrest Matchers

Technically speaking, with Espresso, we do not "find widgets", though it is often simplest to phrase it that way. A more accurate description would be "obtain a `ViewInteraction` object that pertains to a particular widget". The `ViewInteraction` object in turn allows us to perform actions on the underlying widget and check the widget to see if it has a certain state.

For simple widgets — basically, ones that do not involve any sort of collection adapter, like a `ListView` — you can try to get the `ViewInteraction` object via the static `onView()` method on the `Espresso` class. However, the convention is to use a static import for `onView()`:

```
import static android.support.test.espresso.Espresso.onView;
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

This allows you to simply call `onView()` as if it were some global function, rather than having to put `Espresso.` on the front of each occurrence.

The parameter to `onView()` is a `Matcher`. `Matcher` does not come from the Android SDK. Rather, it comes from the Hamcrest project. Another transitive dependency that we get automatically when we pull in Espresso is Hamcrest's matcher library.

There are three main sources of matchers that you can use:

1. `ViewMatcher` contains a number of `static` methods that return matchers that find a `View` with some specific characteristic, such as `withId()` to find a `View` with a particular ID
2. Hamcrest's `Matchers` class has a series of `static` methods that return matchers that help you combine other matchers (e.g., `allOf()` to find a `View` that matches more than one criteria) or work with plain Java collections (e.g., `empty()` to match a collection that is empty)
3. Your own custom matchers, which we will explore later in this chapter

For `ViewMatcher` and `Matchers`, the pattern is to use static imports for their methods as well, such as this import of `withId()`:

```
import static android.support.test.espresso.matcher.ViewMatchers.withId;
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

Here is an Espresso version of the `listCount()` test from the JUnit4 chapter, where we want to validate that a `ListView` contains 25 entries:

```
@Test
public void listCount() {
  onView(withId(android.R.id.list))
    .check(new AdapterCountAssertion(25));
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

Ignoring the `check()` part for now, the `onView(withId(android.R.id.list))` part returns a `ViewInteraction` that is for a `View` whose ID is `android.R.id.list`.

Performing Actions

Given a `ViewInteraction`, one thing that you can do is ask it to `perform()` one or more actions, represented by `ViewAction` objects. The `ViewActions` (note the plural)

class contains a series of `static` methods that create `ViewAction` objects. And, once again, the pattern is to use static imports for those methods.

Here is an Espresso version of the `keyEvents()` test from the JUnit4 chapter, where we want to validate that pressing the down arrow key four times selects the proper row:

```
@Test
public void keyEvents() {
  onView(withId(android.R.id.list))
    .perform(pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN))
    .check(new ListSelectionAssertion(3));
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

Ignoring the `check()` part for now, we retrieve the `ListView` using the same `onView()` code as before. Then, we `perform()` four actions generated by the `pressKey()` method on `ViewActions`:

```
import static android.support.test.espresso.action.ViewActions.pressKey;
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

`pressKey()`, as you might expect, simulates a keypress, given the `KeyEvent`. Other popular actions include:

- `click()` to simulate a click event
- `typeText()` to simulate text entry into an `EditText`
- `scrollTo()` to scroll a `ScrollView` to the point where some view is visible
- `pressImeActionButton()`, to press the action button on the soft keyboard, to trigger whatever action is tied to that button

`perform()` will take care of all synchronization with the main application thread and will ensure that the activity is idle before and after these actions. `perform()` returns the `ViewInteraction`, so you can chain on other operations, such as `check()` calls.

Validating via Assertions… And Possibly More Matchers

Of course, the point behind writing tests is to see if something works or not. That is handled by calling `check()` on a `ViewInteraction`, passing in a `ViewAssertion`

that... well... asserts something. A `ViewAssertion` basically wraps the assertion calls that you might make directly in JUnit4, working with the `ViewInteraction` to confirm that the underlying `View` has some particular state.

Stock Assertions

Compared with the other two phases of writing an Espresso test, there is very little in the way of useful stock assertions.

The `ViewAssertions` class contains a few `static` methods that create `ViewAssertion` objects. The one that you will see most commonly is `matches()`, which asserts that there is now a `View` that matches some supplied matcher, using the same matchers that you might use for `onView()`. This is useful for trivial cases ("is there now a widget whose text is 'Foo Bar'?").

To work with `matches()`, Espresso comes with a number of other matcher implementations that do not match views, but rather match something else, such as:

- `CursorMatchers` returns matchers that match rows in a `Cursor`
- `PreferenceMatchers` returns matchers that match `Preference` objects from a `PreferenceScreen`
- `BoundedMatcher`, from which you can create your own custom matchers

In addition to `ViewAssertions`, Espresso has a few additional classes offering assertions, such as:

- `LayoutAssertions`, for asserting things about how widgets lay out (e.g., confirm these widgets do not overlap)
- `PositionAssertions`... *also* for asserting things about how widgets lay out (e.g., confirm this widget is to the left of this other widget)

Custom Assertions

Since Espresso itself does not provide much in the way of assertions, and since there are few Espresso libraries to help, often you will have to write your own assertions to complete your tests.

This is a matter of writing a class that implements `ViewAssertion` and implements the `check()` method. `check()` receives two parameters:

- the `View` for which you are asserting some state
- a `NoMatchingViewException`, explaining why the first parameter is `null`, if the view could not be found to use with this assertion

Your job in `check()` is to perform ordinary JUnit4-style assertion checks on the `View`, **without modifying the `View`**. This latter part could be tricky, in that you do not necessarily know what does and does not modify the `View`. In general, the "do not modify the `View`" rule is a best-efforts attempt.

With that in mind, let's look at some `ViewAssertion` implementations.

The `listCount()` test method shown above references an `AdapterCountAssertion`:

```
@Test
public void listCount() {
  onView(withId(android.R.id.list))
    .check(new AdapterCountAssertion(25));
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

`AdapterCountAssertion` assumes that it will be given an `AdapterView` and should assert that the underlying count matches a particular value:

```
static class AdapterCountAssertion implements ViewAssertion {
  private final int count;

  AdapterCountAssertion(int count) {
    this.count=count;
  }

  @Override
  public void check(View view,
                    NoMatchingViewException noViewFoundException) {
    Assert.assertTrue(view instanceof AdapterView);
    Assert.assertEquals(count,
      ((AdapterView)view).getAdapter().getCount());
  }
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

The first thing that `check()` should do is assert whether the passed-in `View` is of the appropriate type. Then, it can safely down-cast the `View` as needed and perform the

"real" assertion. In this case, `AdapterCountAssertion` holds onto a `count` from its constructor and compares that to `getCount()` of the adapter in the `AdapterView`.

The `keyEvents()` test method referenced a `ListSelectionAssertion`:

```
@Test
public void keyEvents() {
  onView(withId(android.R.id.list))
    .perform(pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN))
    .check(new ListSelectionAssertion(3));
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

`ListSelectionAssertion` does the same basic thing as does `AdapterCountAssertion`, except it that it validates that the `View` is a `ListView` and compares a known value to `getSelectedItemPosition()`:

```
static class ListSelectionAssertion implements ViewAssertion {
  private final int position;

  ListSelectionAssertion(int position) {
    this.position=position;
  }

  @Override
  public void check(View view,
                    NoMatchingViewException noViewFoundException) {
    Assert.assertTrue(view instanceof ListView);
    Assert.assertEquals(position,
      ((ListView)view).getSelectedItemPosition());
  }
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

The combination of the test methods themselves and these custom assertions is significantly more verbose than the equivalent code using ordinary JUnit4 instrumentation testing:

```
@Test
public void listCount() {
  Assert.assertEquals(25, list.getAdapter().getCount());
```

```
}

@Test
public void keyEvents() {
  sendKeys("4*DPAD_DOWN");
  Assert.assertEquals(4, list.getSelectedItemPosition());
}
```

(from Testing/JUnit4/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityTest.java)

However, custom assertions can be reused, so while Espresso adds a fair bit of overhead to small projects, the savings may add up over larger ones.

The Espresso Test Recorder

The Espresso Test Recorder in Android Studio lets you interact with your app on a device or emulator, while the Recorder makes notes of what you click on and writes skeleton Espresso tests for you. The level of GUI testing that you get is very shallow, but it is exceptionally easy to use.

Starting and Recording

The Run > Record Espresso Test option from the Android Studio main menu will kick off the Recorder, after asking you to choose a device or emulator on which to run the app. Your app is run inside the debugger, and an initially-empty “Record Your Test” window appears:

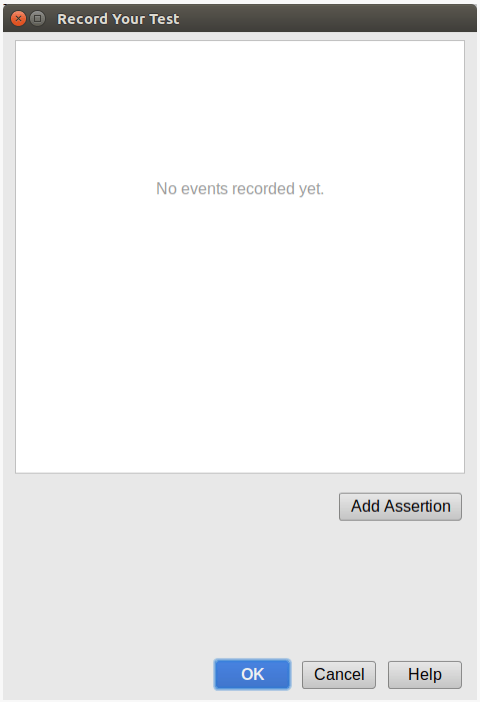


Figure 355: Espresso Test Recorder Events Window, As Initially Launched

As you tap on widgets in your UI, in addition to those taps doing whatever they normally do, events are recorded in that window:

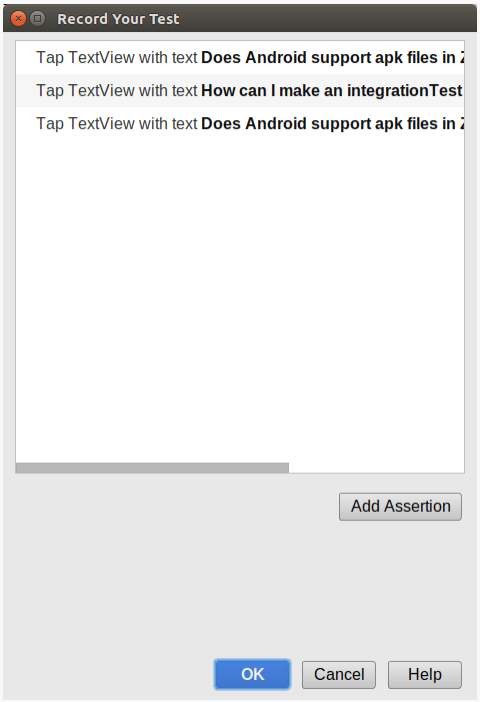


Figure 356: Espresso Test Recorder Events Window, With Some Events

Once you have the app in a state that you want to validate, click the "Add Assertion" button. This captures a screenshot, albeit one that is rotated, if your device or emulator is in landscape mode:

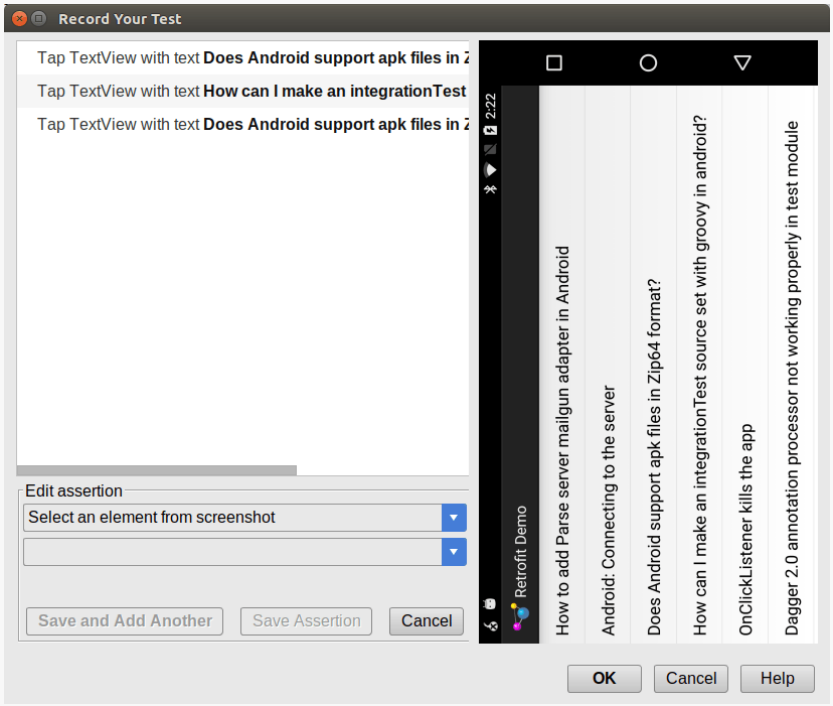


Figure 357: Espresso Test Recorder Events Window, After "Add Assertion" Clicked

A rotated screenshot is unusable; at the present time, the Recorder only really works if your device is in portrait mode.

To add an assertion, you must first click on a widget *in the screenshot* that you want to validate. Alternatively, you can choose the widget from the "Select an element from screenshot" drop-down list.

Then, the second drop-down will allow you to choose what specific assertion you want to apply:

- "text is", to match the text of some `TextView` (or subclass) with its current value
- "exists"
- "does not exist" (which seems odd, considering that it must exist for you to be able to use this dialog)

Clicking "Save and Add Another" adds the assertion and lets you define another one right away:

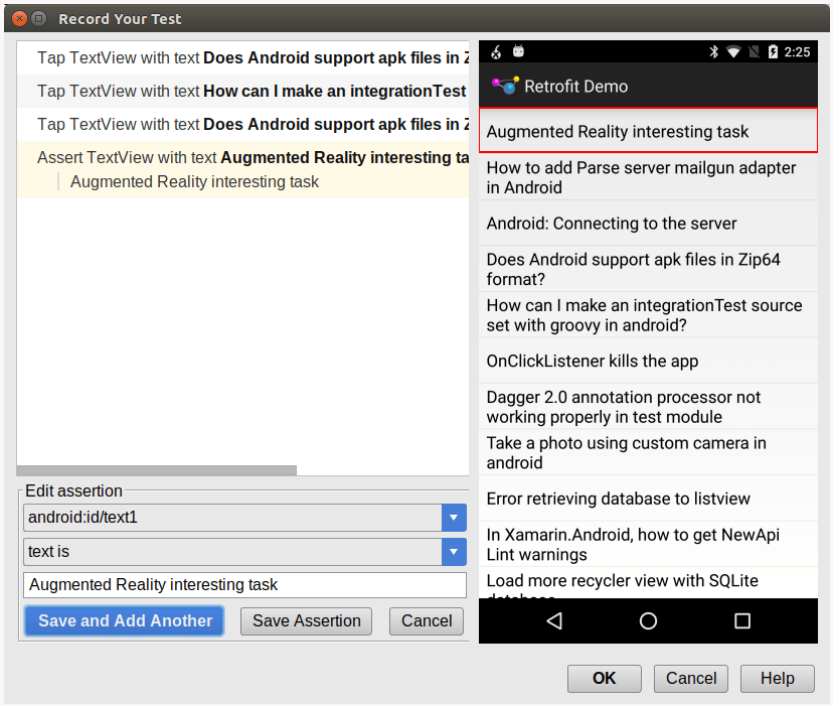


Figure 358: Espresso Test Recorder Events Window, With One Assertion

Clicking "Save Assertion" adds the assertion and returns you to the original screenshot-less rendition of the dialog.

Click OK in the "Record Your Test" dialog to save and apply the recording. You will be prompted for a name to give the JUnit4 test class. Also, if your project is not already set up for Espresso, you will be prompted as to whether or not the Recorder should add Espresso to your Gradle build files.

What You Get

The resulting class will contain:

- An `ActivityTestRule` for the activity that you tested
- A test method, named after the activity (e.g., `mainActivityTest()`) that contains the Espresso code to validate the activity
- Utility methods as needed by the test method

Is This Worthwhile?

Probably not, at least with the edition of the Recorder in Android Studio 2.3.

The Recorder:

- Does not handle `ListView` well
- Supports few types of assertions, as noted above
- Does not support landscape mode while recording, as noted above
- Does not do a very good job of uniquely identifying widgets, resulting in lots of `AmbiguousViewMatcherException` crashes when you run the tests
- And so on

However, this tool may be fine for trivial user interfaces, and the tool may improve in future versions of Android Studio.

Stronger Espresso

You can craft some basic tests using the above techniques, even some not-so-basic tests. However, Espresso offers a fair bit more depth, to tackle more complex testing scenarios.

This chapter does not offer complete coverage of Espresso, skipping many topics like testing `WebView` using the WebDriver Atom system. That being said, here are some more advanced uses of Espresso that you may need in your testing.

Testing AdapterView

`AdapterView` gets complicated because the views that you want to test may or may not exist in the state that you are expecting. Your targeted `ListView` row, for example, may require scrolling and some row recycling before it exists.

Espresso has a slightly different syntax for testing `AdapterView`, to take this into account. Instead of `onView()`, you use `onData()`. Whereas `onView()` takes a `Matcher` that identifies the view to be tested, `onData()` takes a `Matcher` that identifies the specific view state in some item of an `AdapterView` to be tested. However, there are other approaches that one can take to use `onData()`.

In the sample app, `scrollToBottom()` in `DemoActivityRuleTest` tries to confirm that the last position has the proper last word:

```
@Test
public void scrollToBottom() {
  onData(anything())
    .inAdapterView(withId(android.R.id.list))
```

```
      .atPosition(24)
      .check(matches(withText("purus")));
  }
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

The `Matcher` that we provide into `onData()` is `anything()`, which, as the name suggests, matches anything. However, we constrain which item to test via `atPosition(24)`, specifying a particular row in the list based on its position.

`onData()` will work on the first `AdapterView` that it finds, since many activities only have one at most. `inAdapterView()` allows you to identify the specific `AdapterView` that `onData()` should work with.

The `DataInteraction` that `onData()` creates — and is modified by `inAdapterView()` and `atPosition()` — supports the same sort of `check()` semantics as does the `ViewInteraction` returned by `onView()`. Here, we use `matches()` to create our assertion, to confirm that the view identified by the `DataInteraction` has the word that should appear at the end of our list.

Testing RecyclerView

`RecyclerView` is similar enough to `AdapterView` that you might think that you would use `onData()` to work with its contents.

In a word, no.

Instead, you use the same `onView()` that you use for regular widgets. In fact, at the time of this writing, the only thing that Espresso offers specific to `RecyclerView` is `RecyclerViewActions`, which knows how to scroll to a particular item or position, perform actions on items or positions, and so forth.

The `RecyclerViewTest` class tests a `RecyclerView`, from the `MainActivity` in the `rv` sub-package of the sample app. That `MainActivity`, in turn, is cloned from the `ManualDividerList` sample covered in the chapter on `RecyclerView`. It is similar to the `ListView` that we tested earlier in this chapter, showing a vertical-scrolling list of 25 Latin words.

One wrinkle with `MainActivity`, and its `RecyclerViewActivity` base class, is that the `RecyclerView` has no ID. Espresso is *much* easier to use when you have widgets with IDs. There are a couple of ways to get a view with no ID, illustrated in `RecyclerViewTest`.

The `RecyclerView` variant of the `listCount()` test, to confirm that the list has 25 entries, uses one approach to find the `RecyclerView`: the `instanceOf()` method from the standard Hamcrest matchers:

```
@Test
public void listCount() {
  onView(Matchers.<View>instanceOf(RecyclerView.class))
    .check(new AdapterCountAssertion(25));
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/RecyclerViewTest.java)

`instanceOf()` takes the Java class you are seeking as a parameter, and it tries to find an Object that matches that class. However, `instanceOf()` returns a `Matcher` for `Object`, and we need a `Matcher` for `View` to satisfy `onView()`. That is what triggers our need for the complex type-specific call to `instanceOf()`, where we tell `Matchers` that we want the `View`-typed version of `instanceOf()`.

`onView()` with `instanceOf()` gives us a `ViewInteraction` on the `RecyclerView`. Our revised `AdapterCountAssertion` checks the `RecyclerView.Adapter` in the `RecyclerView` to validate the number of items:

```
static class AdapterCountAssertion implements ViewAssertion {
  private final int count;

  AdapterCountAssertion(int count) {
    this.count=count;
  }

  @Override
  public void check(View view,
                    NoMatchingViewException noViewFoundException) {
    Assert.assertTrue(view instanceof RecyclerView);
    Assert.assertEquals(count,
      ((RecyclerView)view).getAdapter().getItemCount());
  }
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/RecyclerViewTest.java)

The `RecyclerView` variant of `scrollToBottom()` is going to simply confirm that we can successfully scroll to position 24 — if the list is shorter than this, we will fail to scroll to that position and have an exception:

```
  @Test
  public void scrollToBottom() {
    onView(withClassName(is(RecyclerView.class.getCanonicalName())))
      .perform(scrollToPosition(24))
      .check(matches(anything()));
  }
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/RecyclerViewTest.java)

This time, to find the `RecyclerView`, we use `withClassName()`. This is a method on `ViewMatchers`, and so it gives us the appropriately-typed `Matcher` for use with `onView()`. However, instead of taking a Java class, it takes a `Matcher` of `String` as a parameter. The `is()` method from the standard Hamcrest matchers returns a `Matcher` that uses `equals()` to compare a supplied value (in this case, the fully-qualified class name for `RecyclerView`). So, `withClassName(is(...))` will find the view of the designated class given the supplied class name.

Once again, `onView()` is returning a `ViewInteraction` on the `RecyclerView`. In `perform()`, we use `scrollToPosition()`, from `RecyclerViewActions`, to scroll the `RecyclerView` to position 24. `RecyclerViewActions` itself is not part of the core Espresso dependency, though. We need to add a dependency on `espresso-contrib` instead, as is shown in the module's `build.gradle` file:

```
apply plugin: 'com.android.application'

dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    androidTestImplementation 'com.android.support:support-annotations:27.0.2'
    androidTestImplementation 'com.android.support:recyclerview-v7:27.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-contrib:3.0.1'
    implementation 'com.android.support:recyclerview-v7:27.0.2'
}

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        applicationId "com.commonsware.android.espresso"
        testApplicationId "com.commonsware.android.espresso.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        testInstrumentationRunnerArguments disableAnalytics: 'true'
    }

    packagingOptions {
        exclude 'LICENSE.txt'
    }
}
```

(from Testing/Espresso/app/build.gradle)

However, `espresso-contrib` depends on an older version of `recyclerview-v7`, so we manually use `androidTestImplementation` to pull in the same version of `recyclerview-v7` that we are using for the "production" code.

Back in `scrollToBottom()`, the `ViewInteraction` created with `onView()` and modified via `perform()` is for the `RecyclerView` itself, not the 25th item. Hence, with this structure, we cannot readily check to see if the 25th item has the proper text. Instead, we settle for ensuring that it matches `anything()`, relying on an exception if for some reason we cannot get to position 24.

Intent Testing

Another optional dependency, `espresso-intents`, allows you to create what amount to mocks and stubs for activities to be started from your code under test. Rather than actually starting those activities, you can intercept the `Intent` that would have been used for `startActivity()` or `startActivityForResult()`, to see if it contains what it should. And, you can provide mock responses to be delivered to `onActivityResult()` for testing `startActivityForResult()` behaviors.

The `Testing/EspressoIntents` sample project is a clone of the `ConfigChange/Bundle` sample app from earlier in the book. It has two buttons, Pick and View. Tapping the Pick button will allow the user to pick a contact out of the list of contacts. Picking a contact then enables the View button, which allows the user to view the selected contact. In instrumentation testing, we want to confirm that the Pick button works as expected.

This app's `build.gradle` file pulls in `espresso-intents` as well as `espresso-core`:

```
apply plugin: 'com.android.application'

dependencies {
    androidTestImplementation 'com.android.support:support-annotations:25.0.3'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:2.2.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-intents:2.2.2'
}

android {
    compileSdkVersion 24
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 14
        targetSdkVersion 22
        testApplicationId "com.commonsware.android.rotation.bundle.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
```

```
        testInstrumentationRunnerArguments disableAnalytics: 'true'
    }

    packagingOptions {
        exclude 'LICENSE.txt'
    }
}
```

(from Testing/EspressoIntents/app/build.gradle)

The `IntentTests` class in the `androidTest` source set contains four test methods, two of which are focused on the pick button and its results. This requires the `IntentTests` class to use a different `@Rule`: `IntentsTestRule`:

```
  @Rule
  public final IntentsTestRule<RotationBundleDemo> main
    =new IntentsTestRule(RotationBundleDemo.class, true);
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

`IntentsTestRule` extends `ActivityTestRule`, so on the whole it behaves the same. However, it has additional hooks for testing `startActivity()` and `startActivityForResult()` with mocks and stubs.

One of the test methods is `canceledPick()`, designed to test what happens if the user presses BACK and exits the contact-picker activity:

```
  @Test
  public void canceledPick() {
    Instrumentation.ActivityResult result=
      new Instrumentation.ActivityResult(Activity.RESULT_CANCELED,
        null);

    intending(hasAction(Intent.ACTION_PICK)).respondWith(result);

    onView(withId(R.id.pick)).perform(click());

    intended(allOf(
      toPackage("com.google.android.contacts"),
      hasAction(Intent.ACTION_PICK),
      hasData(ContactsContract.Contacts.CONTENT_URI)));

    onView(withId(R.id.view)).check(matches(not(isEnabled())));
  }
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

`intending()` is a method that we get from Espresso's `Intents` class. It works with the `IntentsTestRule` to set up a stub with a mock response for our `ACTION_PICK` request that the Pick button will invoke via `startActivityForResult()`. `intending()` takes a `Matcher` of `Intent` objects, identifying which request you are interesting in stubbing. Espresso's `IntentMatchers` class has a series of methods to help you construct an appropriate `Matcher`. In this case, we use `hasAction()` to find our `ACTION_PICK` request. `intending()` returns an oddly-named `OngoingStubbing` class, which represents the stub. On there, we call `respondWith()` to provide the result to feed to `onActivityResult()`. In this case, our response has `RESULT_CANCELED`, simulating the user pressing BACK to exit the contact picker.

We then click the Pick button, by finding it via `onView(withId(R.id.pick))`, then calling `perform()` to `click()` the button. This triggers our "production" code to call `startActivityForResult()`, where our stub delivers the mock response to `onActivityResult()`.

Then, we validate two things:

- First, did we actually send that `Intent`? `intended()` allows us to inspect what `startActivity()` and `startActivityForResult()` calls were made, to validate that it has the appropriate information.
- Second, is the View button still disabled? It starts off disabled, but if the user picks a contact, we enable it. In our case, we did not pick a contact, and so we want to ensure that the button stays disabled, using `matches(not(isEnabled()))`.

The `stubPick()` test method tests the opposite scenario, where the user picks a contact. This time, our result has to have `RESULT_OK` and a sufficiently-valid result `Intent`. In theory, we could have code here that looks up some random contact in `ContactsContract` and uses the `Uri` for it. In this case, all we need is for the result `Intent` to have a `Uri`, so we just use the `ContactsContract.Contacts.CONTENT_URI`:

```
@Test
public void stubPick() {
  Instrumentation.ActivityResult result=
    new Instrumentation.ActivityResult(Activity.RESULT_OK,
      new Intent(null, ContactsContract.Contacts.CONTENT_URI));

  intending(hasAction(Intent.ACTION_PICK)).respondWith(result);

  onView(withId(R.id.pick)).perform(click());
```

```
    intended(allOf(
      toPackage("com.google.android.contacts"),
      hasAction(Intent.ACTION_PICK),
      hasData(ContactsContract.Contacts.CONTENT_URI)));

    onView(withId(R.id.view)).check(matches(isEnabled()));
  }
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

This time, though, the View button should be enabled, so we confirm that as part of our test result validation.

If you are looking to stub a `startActivity()` call, use `respondWith()` and any result. The result will wind up being ignored, but failing to use `respondWith()` does not result in the stub being created, and trying a `null` result crashes Espresso.

Testing Activity Re-Creation and Configuration Changes

Activities get destroyed and re-created by default as a result of a configuration change. You may want to test that process, to ensure that you are retaining the right state information, such as via the `onSavedInstanceState()` `Bundle`. You might even want to test specific configuration changes, such as to confirm that your layouts are set up properly after the user rotates the screen.

While the techniques outlined here can work with plain JUnit4 testing, Espresso simplifies the process a bit. A key challenge with testing this sort of scenario is knowing when the work for the configuration change is done, so you know it is time to go ahead and test the result. Espresso automatically monitors the work queue of the main application thread and only proceeds when the queue indicates that the device is idle, so you know that the configuration change is completed.

Testing the activity destroy-and-create cycle is mostly a matter of calling `recreate()` on the `Activity`. Strictly speaking, `recreate()` is not tied to testing — it is a regular method on `Activity` that you could call whenever. It has limited uses outside of testing, though, which is why you will not run across it very much.

The `recreate()` test method tests `recreate()` to confirm that our View button remains enabled after a destroy-and-recreate cycle:

```
  // inspired by http://stackoverflow.com/a/35139887/115145

  @Test
```

```
public void recreate() {
  stubPick();

  InstrumentationRegistry.getInstrumentation().runOnMainSync(new Runnable() {
    @Override
    public void run() {
      main.getActivity().recreate();
    }
  });

  onView(withId(R.id.view)).check(matches(isEnabled()));
}
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

We first call `stubPick()`, to run through all the code that we tested separately for picking a contact. Then, on the main application thread (via `runOnMainSync()`), we call `recreate()` on the activity, which we get by calling `getActivity()` on our `IntentsTestRule`. Then, we can re-validate the enabled state of the `R.id.view` widget, to confirm that it is still enabled.

Testing a simulated screen rotation is decidedly more complex. Thanks to Chiu-ki Chan, we have a recipe to start with.

First, we need to know what our current orientation is:

```
private int getOrientation() {
  return(InstrumentationRegistry
    .getTargetContext()
    .getResources()
    .getConfiguration()
    .orientation);
}
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

This gets the `orientation` field from the `Configuration` associated with our app under test (`InstrumentationRegistry.getTargetContext()`). That will be one of the `Configuration` values for orientation, such as `Configuration.ORIENTATION_LANDSCAPE`.

To simulate a screen rotation, we can call `setRequestedOrientation()` on the activity. This tells the activity to ignore the actual orientation based on sensors and to use this other orientation instead. Our `rotate()` utility method will flip the orientation from whatever it actually is (via `getOrientation()`) to the opposite:

```
private void rotate() {
  int target=
    (getOrientation()==Configuration.ORIENTATION_LANDSCAPE ?
      ActivityInfo.SCREEN_ORIENTATION_PORTRAIT :
      ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);

  main.getActivity().setRequestedOrientation(target);
}
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

We also have a `testOrientation()` helper method:

```
private int testOrientation() {
  int orientation=getOrientation();

  if (orientation==Configuration.ORIENTATION_LANDSCAPE) {
    onView(withId(R.id.content))
      .check(new OrientationAssertion(LinearLayout.HORIZONTAL));
  }
  else {
    onView(withId(R.id.content))
      .check(new OrientationAssertion(LinearLayout.VERTICAL));
  }

  return(orientation);
}
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

This finds our current orientation (via `getOrientation()`). Based upon that, it validates whether our `LinearLayout` has the correct orientation. To aid with this, the layouts now have an ID for the `LinearLayout` (`R.id.content`), so we do not have to go through messy code to try to find the right object. There is no built-in assertion for testing the orientation of a `LinearLayout`, but it is easy enough for us to write our own `OrientationAssertion`:

```
static class OrientationAssertion implements ViewAssertion {
  private final int orientation;

  OrientationAssertion(int orientation) {
    this.orientation=orientation;
  }

  @Override
  public void check(View view,
                    NoMatchingViewException noViewFoundException) {
```

```
    Assert.assertTrue(view instanceof LinearLayout);
    Assert.assertEquals(orientation,
      ((LinearLayout)view).getOrientation());
  }
}
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

All that gets used by the `orientation()` `@Test` method:

```
@Test
public void orientation() {
  int original=testOrientation();

  rotate();

  int postRotate=testOrientation();

  Assert.assertFalse("orientation changed", original==postRotate);
}
```

(from Testing/EspressoIntents/app/src/androidTest/java/com/commonsware/android/rotation/bundle/IntentTests.java)

Here we:

- Test our current orientation, noting what orientation that is
- Simulate rotating the screen
- Test our new orientation, noting what orientation that is
- Validate that the orientation did actually change, using an ordinary JUnit4 assertion

Custom Matchers

Espresso has a robust set of `Matcher` implementations. Most likely, you will not need to create your own custom matcher. This is particularly true if you do a good job of putting `android:id` values on your widgets, so you can just use `withId()`.

That being said, it is certainly possible to create your own `Matcher`, if you really want to.

As Chiu-ki Chan points out, the key is to use `BoundedMatcher` as a base class. This creates a `Matcher` for some type (e.g., `View`) and subtype (e.g., `TextView`). The dual-typing appears to be due to some odd interactions between Espresso, Hamcrest, and Java generics.

The Testing/EspressoMatcher sample project is a clone of another sample app, this one using TextInputLayout from the Design Support library, covered elsewhere in the book. TextInputLayout is a wrapper around an EditText that provides the "floating label" pattern.

The actual "floating label" of a TextInputLayout comes from its hint. However, while Espresso has a withHint() matcher method, that is for TextView and subclasses. TextInputLayout does not inherit from TextView; it has its own clone of the hint-related methods (e.g., setHint()). If we want to find a TextInputLayout by hint, we need our own matcher.

The TILTest class in androidTest implements such a TILHintMatcher:

```
private static class TILHintMatcher
  extends BoundedMatcher<View, TextInputLayout> {
  private final Matcher<CharSequence> textMatcher;

  TILHintMatcher(Matcher<CharSequence> textMatcher) {
    super(TextInputLayout.class);

    this.textMatcher=textMatcher;
  }

  @Override
  protected boolean matchesSafely(TextInputLayout item) {
    return(textMatcher.matches(item.getHint()));
  }

  @Override
  public void describeTo(Description description) {
    description.appendText("with hint: ");
    textMatcher.describeTo(description);
  }
}
```

(from Testing/EspressoMatcher/app/src/androidTest/java/com/commonsware/android/design/til/TILTest.java)

TILHintMatcher is a subclass of BoundedMatcher, declaring that it is creating a Matcher of View, but where *really* the View should be a TextInputLayout. We use a Matcher of CharSequence as our identifier — we will use that to compare against the hint to see if a given TextInputLayout is the one that we want.

There are two methods that you need to implement on your BoundedMatcher subclass. The big one is matchesSafely(). This gives you an instance of your specific type (in this case, TextInputLayout). Your job is to return true if this widget

matches the rules for this matcher instance. In our case, we confirm that the hint of the `TextInputLayout` satisfies our `CharSequence` `Matcher`.

The other method is `describeTo()`. This builds up a description of the matcher, used for diagnostic purposes, such as in cases where more than one `View` matches. The pattern is to update the supplied `Description` object with details about your matching rule. Here, we indicate that we are checking to see if the hint matches the `textMatcher` rule.

The activity under test — `LaunchDemo` — has a `TextInputLayout` wrapped around a `TextInputEditText`, where the user can type in a URL. There is also a `browse` button that, when clicked, will open an `ACTION_VIEW` activity on that URL. We want to confirm that this works, using the `Intent` stub-and-mock approach outlined earlier in this chapter.

In this case, we need to:

- Enter some text into the `TextInputEditText`, then
- Click the `browse` button

This would be simplest if we just gave a unique ID to the `TextInputEditText`, but that will not test out a custom matcher. So, we go through a more complex approach, involving our `TILHintMatcher`:

```
@Test
public void til() {
  onView(allOf(withParent(withTILHint("URL")),
    Matchers.<View>instanceOf(TextInputEditText.class)))
    .perform(typeText(URL), closeSoftKeyboard());

  Instrumentation.ActivityResult result=
    new Instrumentation.ActivityResult(Activity.RESULT_CANCELED,
      null);

  intending(hasAction(Intent.ACTION_VIEW)).respondWith(result);

  onView(withId(R.id.browse)).perform(click());

  intended(allOf(hasAction(Intent.ACTION_VIEW), hasData(URL)));
}

private Matcher<View> withTILHint(CharSequence text) {
  return(new TILHintMatcher(is(text)));
}
```

(from Testing/EspressoMatcher/app/src/androidTest/java/com/commonsware/android/design/til/TILTest.java)

It is possible that we could have several `TextInputEditText` widgets in our layout. In truth, we do not. But, in case we elect to add more later, we do not want to simply look for an instance of `TextInputEditText`, as that may not be unique. Instead, we want to find the one that is a child of the `TextInputLayout` that has a hint of `"URL"`.

`withTILHint()` is a utility method that simply wraps a supplied string in an `is()` matcher, then creates a `TILHintMatcher` wrapped around that.

`withParent(withTILHint("URL"))`, therefore, is a matcher that matches any widget that has a parent that is a `TextInputLayout`, with a hint of `"URL"`.

You might think that would be sufficient, given our layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me">

    <requestFocus/>
  </Button>

  <android.support.design.widget.TextInputLayout
    android:id="@+id/til"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <android.support.design.widget.TextInputEditText
      android:id="@+id/url"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/url"
      android:inputType="textUri"/>
  </android.support.design.widget.TextInputLayout>

</LinearLayout>
```

(from Testing/EspressoMatcher/app/src/main/res/layout/main.xml)

As it turns out, `TextInputLayout` has its own internal structure. `onView(withParent(withTILHint("URL")))` will fail, indicating that more than one `View` matches.

So, we use Hamcrest's `allOf()` method, which creates a matcher that uses a boolean AND operation on all supplied matchers — all have to match for `allOf()` to consider it a match. So, we also use `instanceOf()` to constrain us to `TextInputEditText` widgets. That, finally, gives us our `TextInputEditText` widget, and makes us glad that we can use `android:id` in our own code and avoid all of this hassle.

To complete our testing, we call `perform()` to perform some actions on the `TextInputEditText`. `perform()` can execute any number of actions, and here we are performing two:

1. `typeText()`, to type in a URL (identified here as a `URL` constant)
2. `closeSoftKeyboard()`, to ensure that the soft keyboard has collapsed after testing

If we did not `perform()` `closeSoftKeyboard()`, the tests would work in portrait mode, but not in landscape, given the way that the full-screen landscape input method editor works.

We then:

- Set up to stub the `ACTION_VIEW` `Intent` request, with a throwaway response
- `click()` the `browse` button
- Validate that we did invoke an `ACTION_VIEW` `Intent` with the proper URL

Opting Out of Analytics

All of the `build.gradle` files shown in this chapter have the following line in `defaultConfig`:

```
testInstrumentationRunnerArguments disableAnalytics: 'true'
```

By default, your Espresso tests send data about your tests to Google. This line passes arguments to the test runner that disable these analytics.

Waiting for the World to Change

Our activities often trigger asynchronous work: loading data from a database, loading content from a `ContentProvider`, executing a Web service call, etc. Sometimes, that work is triggered by the start of the activity. Sometimes, that work is triggered by UI events.

Usually, our tests need to wait for that work to complete before we can proceed with confirming the results. For example, if tapping an action bar item refreshes the `RecyclerView` contents via some asynchronous work, we cannot determine whether or not the refresh worked until that asynchronous operation ends.

Even simpler things would seem to need more synchronization than we are writing in our tests. Let's go back to the `keyEvents()` test from earlier in this chapter:

```
@Test
public void keyEvents() {
  onView(withId(android.R.id.list))
    .perform(pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN),
      pressKey(KeyEvent.KEYCODE_DPAD_DOWN))
    .check(new ListSelectionAssertion(3));
}
```

(from Testing/Espresso/app/src/androidTest/java/com/commonsware/android/abf/test/DemoActivityRuleTest.java)

We simulate four down-arrow presses, then check to see if the proper list row is selected.

However, `keyEvents()` runs on a background thread, not the main application thread. In theory, there is a race condition here: will the work associated with those four down-arrow events be completed by the time we go to check the selection state of the `ListView`?

It turns out that Espresso handles this automatically. It waits until the work queue for the main application thread shows that there is no more work ready to process. Then, and only then, will the `check()` logic be applied.

Espresso also waits on select other things, notably `AsyncTasks`. And, through an `IdlingResource`, we can teach it to wait for other asynchronous work: arbitrary threads, an `IntentService`, and so on.

What's an IdlingResource?

An `IdlingResource` is an interface. Implementations of it know how to monitor some background work for completion. You can register an `IdlingResource` with Espresso via `Espresso.registerIdlingResources()`, later removing it via `Espresso.unregisterIdlingResources()`.

An `IdlingResource` needs to do two main things:

1. Return whether the resource being monitored is idle at the moment, via an `isIdleNow()` method
2. Track a `ResourceCallback` instance and call an `onTransitionToIdle()` method on it when the resource becomes idle

Using an IdlingResource

In some cases, you will be able to use a pre-built `IdlingResource`.

Espresso itself comes with a `CountingIdlingResource` that you can use a bit like a `CountDownLatch`, calling `increment()` when work is added and `decrement()` when work is completed. When the counter falls to zero from a non-zero value, the `CountingIdlingResource` will call `onTransitionToIdle()` on its `ResourceCallback`. And, `isIdleNow()` is simply based on the counter.

Sometimes, you will find existing implementations from third parties. For example, Jake Wharton has written an `OkHttp3IdlingResource` for use with OkHttp3. The `Testing/EspressoIdle` sample project demonstrates its use.

The activity under test is a variation on the OkHttp3 "show the latest Stack Overflow questions" sample from the chapter on Internet access. That activity (`MainActivity`) holds an `OkHttpClient` instance to be used by its fragment, returned via a `getOkHttpClient()` method:

```
package com.commonsware.android.okhttp;

import android.support.v4.app.FragmentActivity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import okhttp3.OkHttpClient;

public class MainActivity extends FragmentActivity
  implements QuestionsFragment.Contract {
  private final OkHttpClient client=
```

```
    new OkHttpClient.Builder().build();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
      getSupportFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new QuestionsFragment()).commit();
    }
  }

  @Override
  public void onQuestion(Item question) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(question.link)));
  }

  OkHttpClient getOkHttpClient() {
    return(client);
  }
}
```

(from Testing/EspressoIdle/app/src/main/java/com/commonsware/android/okhttp/MainActivity.java)

`QuestionsFragment` then uses OkHttp3 to request the latest 100 Stack Overflow questions, parsing the JSON response with Gson, and loading them into the fragment's `ListView`. This is handled in the `onViewCreated()` method:

```
@Override
public void onViewCreated(final View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  OkHttpClient client=((MainActivity)getActivity()).getOkHttpClient();
  Request request=new Request.Builder().url(SO_URL).build();

  client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
      Log.e(getClass().getSimpleName(), "Exception loading JSON", e);
    }

    @Override
    public void onResponse(Call call, Response response)
      throws IOException {
      if (response.isSuccessful()) {
        Reader in=response.body().charStream();
        BufferedReader reader=new BufferedReader(in);
        final SOQuestions questions=
          new Gson().fromJson(reader, SOQuestions.class);
```

```
        reader.close();

        view.post(new Runnable() {
          @Override
          public void run() {
            setListAdapter(new ItemsAdapter(questions.items));
          }
        });
      }
      else {
        Log.e(getClass().getSimpleName(), response.toString());
      }
    }
  });
}
```

(from Testing/EspressoIdle/app/src/main/java/com/commonsware/android/okhttp/QuestionsFragment.java)

Note that `onViewCreated()` uses `enqueue()`, rather than `execute()`, so the HTTP request is performed on an OkHttp3-supplied background thread. We do not know if this is an `AsyncTask` or some other type of thread, and so we do not know for certain if Espresso will know to wait until this request is complete. Hence, in our tests, we really should use an `IdlingResource` to confirm that the asynchronous work is completed before seeing if the results match our expectations.

The `OkHttpTests` instrumentation test class uses an `ActivityRule`, named `main`, to set up our activity under test. The `moreReliableAsyncTest()` method then uses `OkHttp3IdlingResource` to help out our test code:

```
@Test
public void moreReliableAsyncTest() {
  IdlingResource idleWild=
    OkHttp3IdlingResource.create("okhttp3",
      main.getActivity().getOkHttpClient());

  Espresso.registerIdlingResources(idleWild);

  try {
    onView(withId(android.R.id.list))
      .check(new AdapterCountAssertion(100));
  }
  finally {
    Espresso.unregisterIdlingResources(idleWild);
  }
}
```

(from Testing/EspressoIdle/app/src/androidTest/java/com/commonsware/android/okhttp/OkHttpTests.java)

An `IdlingResource` has a name, which needs to be unique among registered `IdlingResource` instances. So, we use `okhttp3`. `OkHttp3IdlingResource` also needs our `OkHttpClient`, which we can get by retrieving it from the `MainActivity`, which is available to us courtesy of the `main` `ActivityRule`.

We then:

- Call `registerIdlingResources()` to register our `OkHttp3IdlingResource`
- Perform our tests, to confirm that we got 100 items in the list
- Call `unregisterIdlingResource()`, as we can no longer use our `OkHttp3IdlingResource` once the activity is destroyed; any new activity instances will have their own `OkHttpClient` in our implementation

Implementing a Custom IdlingResource

A custom `IdlingResource` can be simple or complex, depending on how difficult it is to determine whether a resource is idling and when it starts idling.

In the case of `OkHttp3IdlingResource`, OkHttp3 itself exposes a API, in the form of a `Dispatcher` object, that provides an API ideal for an `IdlingResource`:

- A `Dispatcher` has `runningCallsCount()` for use by `isIdleNow()`
- A `Dispatcher` has `setIdleCallback()`, to be called when the running calls count drops to zero

In this case, the reason for the clean integration may be tied to the fact that Jake Wharton works on OkHttp3.

Sometimes, you have to use a less elegant approach, because the resource you wish to monitor does not offer an appropriate monitoring API. Chiu-ki Chan wrote an `IntentServiceIdlingResource` that uses `ActivityManager` to watch for when a specific `IntentService` implementation is no longer listed as a running service. There is no way to register a callback with Android to find out when an arbitrary service is destroyed, so she only invokes the `ResourceCallback` `onTransitionToIdle()` method as part of `isIdleNow()` processing.

Testing with UI Automator

Yet another approach for testing Android applications is UI Automator. This is designed for integration testing, both how your app components integrate with one another (e.g., activities starting activities) and how your app components integrate with the rest of a device, including other applications.

In early 2015, Google released version 2.0 of the UI Automator framework. This update ties UI Automator into the same instrumentation testing engine that is used for JUnit4 testing. This also makes it possible to run UI Automator tests through Android Studio and the Android Gradle Plugin, which previously had been difficult.

Prerequisites

This chapter assumes that you have read the chapter on JUnit4.

What Is UI Automator?

UI Automator, as the name suggests, automates UIs. It simulates user input, in the form of tapping on items and the like. It does so without modifying your process' contents. Tests run by UI Automator are implemented in JUnit, and those tests have limited access to the widgets inside of a UI. Such access not only allows for directing simulated user input (e.g., "click the OK button"), but also for asserting that various test conditions are true (e.g., "does the list have five rows?"). In this respect, UI Automator behaves like traditional Android JUnit testing.

Why Choose UI Automator Over Alternatives?

In some respects, UI Automator represents the worst of both worlds. You have to use JUnit, making test authoring a challenge for those not skilled with Java. Yet you only have fairly generic access to an activity's widgets, versus the complete white-box capability of normal instrumentation-based JUnit testing.

Hence, why would anyone bother?

The big thing that UI Automator offers over classic JUnit testing is greater ability to test an *application* versus testing *individual components*. The classic JUnit test cases are organized around testing some specific component, such as using `ActivityInstrumentationTestCase2` to exercise some specific activity. Testing the flow of work between activities is difficult from classic JUnit, but is relatively easy with UI Automator. You can also use this for integration testing, as you can exercise and analyze applications other than your own, such as to confirm that you are starting a third-party app correctly.

Similarly, classic JUnit testing cooks up activity instances "out of thin air". Instead, UI Automator executes normal UI operations to create the activities, such as tapping on your app's icon in the home screen launcher. This more accurately simulates what a user will do — users are far more likely to tap on a launcher than to hack into your Dalvik VM and manually instantiate an activity.

You can see a set of UI Automator tests in a suitable project in the `Testing/UiAutomator` directory. Note, though, that the UI Automator tests will only work successfully on Android 4.x emulators, and perhaps a few other environments. The tests are testing the integration of the home screen to the app, along with the app's functionality, and the particular code used to navigate the home screen will only work with the stock Android home screen, not necessarily any manufacturer's home screen or third-party home screen.

Gradle and Android Studio Settings

Your project needs to be set up to use the `AndroidJUnitRunner` as is outlined in the chapter on JUnit4.

For UI Automator, you additionally need to have an `androidTestImplementation` dependency on the `uiautomator-v18` artifact:

```
dependencies {
    androidTestCompile 'com.android.support.test:rules:0.3'
    androidTestCompile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.0'
}
```

(from Testing/UiAutomator/app/build.gradle)

Here the `-v18` suffix, as with the regular Android Support package libraries, means that this only works on API Level 18 and higher.

If you wish to run the tests from Android Studio, you will also need to set up a run configuration, as outlined in the chapter on JUnit4.

Creating a Test Case

Your test case classes do not need to inherit from any particular base class, just like regular JUnit4 tests. They do need to be annotated with the `@RunWith(AndroidJUnit4.class)` annotation:

```
@RunWith(AndroidJUnit4.class)
public class ListTests {
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Your test case is welcome to have `@Before`, `@After`, and other setup/teardown methods, in addition to `@Test` methods, just like a regular JUnit4 test case. In fact, from Android's standpoint, UI Automator tests *are* just regular JUnit4 test cases — you are welcome to have UI Automator test cases and regular instrumentation testing JUnit4 test cases in the same `androidTest` source set.

Performing Device-Level Actions

The root of most of our work with UI Automator is a `UiDevice` object. This allows us to perform device-level actions, such as pressing BACK or HOME.

To get a `UiDevice`, call the static `getInstance()` method on `UiDevice`, passing in the `Instrumentation` that you get from `InstrumentationRegistry.getInstrumentation()`:

```
  @Before
  public void setUp() throws UiObjectNotFoundException {
    device=UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());
    openActivity();
  }
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Here, we get the `UiDevice` and stash it in a data member for the life of this `ListTests` instance.

`UiDevice` has many methods that allow you to perform device-level actions, such as calling `pressHome()` to press the HOME button (and thereby bring up the home screen). Similarly, you can call:

- `pressBack()` and `pressMenu()` for the BACK and MENU buttons
- `pressDPadUp()`, `pressDPadLeft()`, etc. for D-pad events
- `pressRecentApps()` to bring up the recent tasks list
- `pressKeyCode()` to press an arbitrary key based on the keycode from `KeyEvent`

...and so on.

Inspecting and Interacting with the UI

Of course, pressing some buttons is not especially useful on its own, only as a means to an end, such as launching your activity. To do more than this, you will need to get your hands on widgets and containers, to perform operations related to them.

The key is that you can "get your hands on widgets and containers" *from whatever activity is in the foreground*. This is not limited to your own app, but rather works for *any* app, including the home screen itself.

The following sections will work through some common UI Automator operations, in the context of the `openActivity()` from the `ListTests` class in the sample project. This method, called from `setUp()`, consolidates the work to bring an instance of our production activity to the foreground, by means of interacting with the home screen:

```
private void openActivity() throws UiObjectNotFoundException {
  device.pressHome();

  UiObject allAppsButton=
      device.findObject(new UiSelector().description("Apps"));

  allAppsButton.clickAndWaitForNewWindow();

  UiObject appsTab=device.findObject(new UiSelector().text("Apps"));

  appsTab.click();
```

```
    UiScrollable appViews=
        new UiScrollable(new UiSelector().scrollable(true));

    appViews.setAsHorizontalList();

    UiObject ourApp=
        appViews.getChildByText(new UiSelector().className("android.widget.TextView"),
                                "Action Bar Fragment Demo");

    ourApp.clickAndWaitForNewWindow();

    UiObject appValidation=
        device.findObject(new UiSelector().packageName("com.commonsware.android.abf"));

    Assert.assertTrue("Could not open test app", appValidation.exists());
  }
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Finding and Interacting with Widgets

`openActivity()` starts by calling `pressHome()` on the `UiDevice`, to ensure that the home screen is in the foreground:

```
    device.pressHome();
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Next, we want to bring up the home screen's launcher, showing the available launchable activities, so that we can find our app and launch it. What a user would do, on a stock Android environment like an emulator, would be to click on the appropriate button to bring up the launcher. We need to do the same thing, except from our test code. This implies:

- Finding that widget
- Simulating a click of that widget

Web developers are used to finding DOM nodes by CSS queries. Developers using XML are used to using XPath queries to find particular elements. Along the same lines, UI Automator gives us a flexible system to find widgets in the foreground activity, by means of a `UiSelector` object, typically created using the public zero-argument constructor (i.e., `new UiSelector()`).

In CSS, a "selector" can identify DOM nodes by class, `id`, or ones with particular properties. A `UiSelector` can do much the same thing. So, the first `UiSelector` created in `openActivity()` will find a widget in the foreground activity whose "description" is `Apps` (`new UiSelector().description("Apps")`). Here, "description"

will mean either the text of a `TextView` or the `android:contentDescription` of other types of widgets.

How do we know that this particular button has a "description" of `Apps`? In this case, we found out using `uiautomatorviewer`, which will be discussed in a future update to this chapter.

By passing our `UiSelector` to `findObject()` on the `UiDevice`, we get a `UiObject` that, hopefully, knows how to interact with this particular button of the home screen. In particular, we call `clickAndWaitForNewWindow()` on it, which taps the button and blocks until something else (e.g., a new activity) has taken over the foreground:

```
    UiObject allAppsButton=
        device.findObject(new UiSelector().description("Apps"));

    allAppsButton.clickAndWaitForNewWindow();
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

The stock Android launcher has two tabs, one for apps and one for (app) widgets. We need to ensure that the apps tab is selected. So, once again, we create a `UiSelector` and use it to create a `UiObject` to represent the apps tab. This time, we use `text()` instead of `description()`. `text()` will find a widget based solely on its display text (e.g., `android:text` of a `TextView`). In truth, we could have used `description()` here as well, with the same results.

Then, we call `click()` on the `UiObject`, to simulate a tap on this tab, to ensure that is the selected tab.

```
    UiObject appsTab=device.findObject(new UiSelector().text("Apps"));

    appsTab.click();
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Dealing with Collections

Finding widgets by text or description is fairly easy when there is only one possible widget that has that text or description. Things get more complicated when you are dealing with a collection of widgets, such as an `AdapterView`.

For example, the Apps tab of the standard Android launcher uses a `GridView` to show up to 20 launchable activities. Then, you need to swipe horizontally, courtesy of a `ViewPager`, to uncover additional `GridView` collections of launchable activities.

A `UiCollection` helps deal with this, in terms of allowing you to inspect a collection of widgets, including performing the necessary swipe operations to access all of the contents.

A `UiSelector` called with `scrollable(true)` will return a widget that is scrollable. Creating a `UiCollection` with that `UiSelector` will create a `UiCollection` around the first scrollable widget. In the case of the Apps tab, that will be the `ViewPager`-and-`GridView` combination.

In our case, to get to other elements in the collection, you need to swipe horizontally. To configure the `UiCollection` that way, we have to call `setAsHorizontalList()` on the `UiCollection`:

```
UiScrollable appViews=
    new UiScrollable(new UiSelector().scrollable(true));

appViews.setAsHorizontalList();

UiObject ourApp=
    appViews.getChildByText(new UiSelector().className("android.widget.TextView"),
                            "Action Bar Fragment Demo");
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Finding Widgets By Type

In that collection, we want to find the item that contains our app's caption. This test project is designed to test the same sample app that was tested in the JUnit chapter, a slightly modified version of an early action bar sample. Our launcher entry's name will be "Action Bar Fragment Demo", as that is what we set up in the production project's manifest and string resources. So, we need to find the entry in the `ViewPager`-of-`GridViews` that has that title.

To do that, we will create yet another `UiSelector`. This time, though, we will find widgets by type, specifying `className("android.widget.TextView")` to only work with `TextView` widgets.

That `UiSelector` is passed into the `getChildByText()` method of `UiCollection`, which will iterate over the children to find the first one that matches the `UiSelector` and where the selected widget contains the supplied text:

```
    UiObject ourApp=
        appViews.getChildByText(new UiSelector().className("android.widget.TextView"),
                                "Action Bar Fragment Demo");
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Then, we again call `clickAndWaitForNewWindow()`, to tap on our launcher entry, triggering our app's activity to come to the foreground:

```
    ourApp.clickAndWaitForNewWindow();
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Asserting Conditions

`UiSelector` and `UiObject` can also be used for some operations that do not fit the normal widgets-and-containers pattern shown above.

For example, now that we have opened a window from our app to be tested, it would be nice to confirm that, indeed, this is our app, and that our `openActivity()` method did not open some other app by mistake.

To do this, we can create a `UiSelector` and apply `packageName()`, to constrain the selection to widgets coming from an app with our desired package name:

```
    UiObject appValidation=
        device.findObject(new UiSelector().packageName("com.commonsware.android.abf"));
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

The `UiObject` we create always exists (i.e., is not `null`), as we are creating it via the constructor. However, it is entirely possible that our `UiSelector` cannot match any widget, such as would be the case if we accidentally opened the wrong app and tried to find a widget stemming from our package. The `exists()` method on a `UiObject` returns `true` if the `UiObject` is pointing at an actual widget, `false` otherwise. Hence, we can assert that we indeed have a widget coming from our package:

```
    Assert.assertTrue("Could not open test app", appValidation.exists());
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

The net result is that we open our main activity and confirm that, indeed, that is what we opened.

And Now… The Real Test Methods

All of that was just to get the activity for testing onto the screen.

Now the *real* testing begins.

The `ListTests` class has two test methods, `testContents()` and `testAdd()`, designed to (lightly) exercise the UI.

testContents()

The objective of the `testContents()` method is to confirm that the 25 words all appear in the `ListView`.

To do that, we:

- Create a `UiScrollable` for a `UiSelector` that finds the `ListView` in our activity
- Mark that `UiScrollable` as being a vertical list, where swipes up and down will expose the various children
- Iterate over the array of words, finding the `TextView` for each word and confirming that this widget does indeed exist

```
@Test
public void testContents() throws UiObjectNotFoundException {
  UiScrollable words=
      new UiScrollable(
                      new UiSelector().className("android.widget.ListView"));

  words.setAsVerticalList();

  for (String s : items) {
    Assert.assertNotNull("Could not find " + s,
        words.getChildByText(new UiSelector().className("android.widget.TextView"),
            s));
  }
}
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

testAdd()

The objective of the `testAdd()` method is to add a new word to the list, via the `EditText` widget in our action bar, then confirm that the new word was actually added to the list.

To do that, we:

- Retrieve the `EditText` by finding the widget whose text is "Word" (the hint of our `EditText`)
- Call `setText()` to fill in `snicklefritz` into the `EditText` widget, which `UiObject` accomplishes by actually typing in the value
- Call `pressEnter()` on the `UiDevice` to simulate pressing the Enter key of a keyboard, which will trigger our action listener in the test activity and will add the word to the list
- Create a `UiScrollable` for a `UiSelector` that finds the `ListView` in our activity
- Mark that `UiScrollable` as being a vertical list, where swipes up and down will expose the various children
- Try to find a `TextView` whose text is `snicklefritz` and assert that it was found

```
@Test
public void testAdd() throws UiObjectNotFoundException {
  UiObject add=device.findObject(new UiSelector().text("Word"));

  add.setText("snicklefritz");
  device.pressEnter();

  UiScrollable words=
      new UiScrollable(
                      new UiSelector().className("android.widget.ListView"));

  words.setAsVerticalList();

  Assert.assertNotNull("Could not find snicklefritz",
      words.getChildByText(new UiSelector().className("android.widget.TextView"),
          "snicklefritz"));
}
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Cleaning Up

Our `ListTests` class also has a `tearDown()` method, invoked by JUnit after each test method courtesy of the `@After` annotation. Here, we press BACK twice, to return us to the main home screen from our activity, setting things back up for the next test method:

```
@After
public void tearDown() {
  device.pressBack();
  device.pressBack();
}
```

(from Testing/UiAutomator/app/src/androidTest/java/com/commonsware/android/abf/uiautomator/ListTests.java)

Running Your Tests

You run your UI Automator tests as you would any other instrumentation test:

- By running the run configuration that you set up for your tests in Android Studio
- By running commands like `gradle connectedCheck` at the command line
- Through integrations into your continuous integration server or similar build infrastructure

Finding Your Widgets

The key to finding your desired widgets stems in large part from the `text()` or `description()` methods on `UiSelector`. Of those two, the latter is more flexible, as it will use the `android:contentDescription` from any widget, while `text()` is limited to `TextView` and its subclasses.

However, this implies that your widgets have `android:contentDescription` defined. This is also important for accessibility, and therefore is a good idea regardless of its use with UI Automator.

For testing your own code, you can also find widgets via their resource IDs. `UiSelector` has `resourceId()` and `resourceIdMatches()` methods to configure the resource ID you want. As the `resourceIdMatches()` method name suggests, the resource ID here is a *string representation* of the resource name. It will be of the form `your.app.package:id/resource` (e.g., `com.commonsware.android.hotkey:id/editor`). However:

- Note that this requires API Level 18 or higher versions of the two JARs (`UI Automator.jar` and `android.jar`)
- Note that this requires running the tests on an API Level 18+ device or emulator
- Bear in mind that third party apps are welcome to rename their widgets when they wish, so your integration tests may break when third parties do so

Using the UI Automator Viewer

Identifying widgets can be a bit tricky with UI Automator. Identifying widgets *in other apps*, for your integration tests, would in theory be next to impossible. After all, while the Android Studio layout inspector and the older Hierarchy View tools can give you widget IDs, that only works with debuggable apps. *Your* app may be debuggable, but the app you are trying to integrate with probably is not.

Fortunately, we have the UI Automator Viewer. This tool basically walks the view hierarchy of whatever activity is in the foreground of a device (or emulator) and gives us access to whatever information is exposed by the accessibility APIs. Nowadays, this includes widget IDs, in addition to more traditional accessibility data like the text in a `TextView`, the `contentDescription` of an `ImageView`, and so on.

At the present time, the UI Automator Viewer is not integrated into Android Studio or the Android Device Monitor GUIs. Instead, you will have to launch it the old-fashioned way, by running the `uiautomatorviewer` command from the command line. This will map to a batch file or shell script in the `tools/` directory of your Android SDK installation.

When initially launched, the UI Automator Viewer does not look like much:

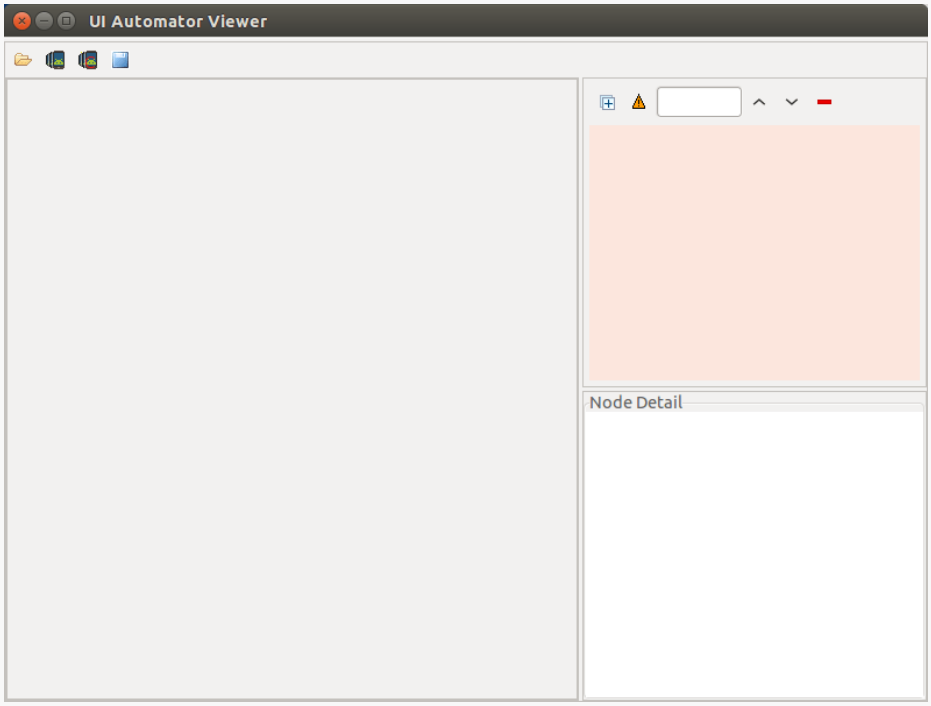


Figure 359: UI Automator Viewer, As Initially Launched

Given that you have a device or emulator ready, you can click the second icon from the left in the toolbar, to capture the view hierarchy of the foreground activity. This will give you:

- A screenshot of the foreground activity in the main area of the UI Automator Viewer screen
- The view hierarchy of that activity, in the upper-right corner of the UI Automator Viewer screen
- Properties of a node from the selected view, in the lower-right corner of the UI Automator Viewer screen

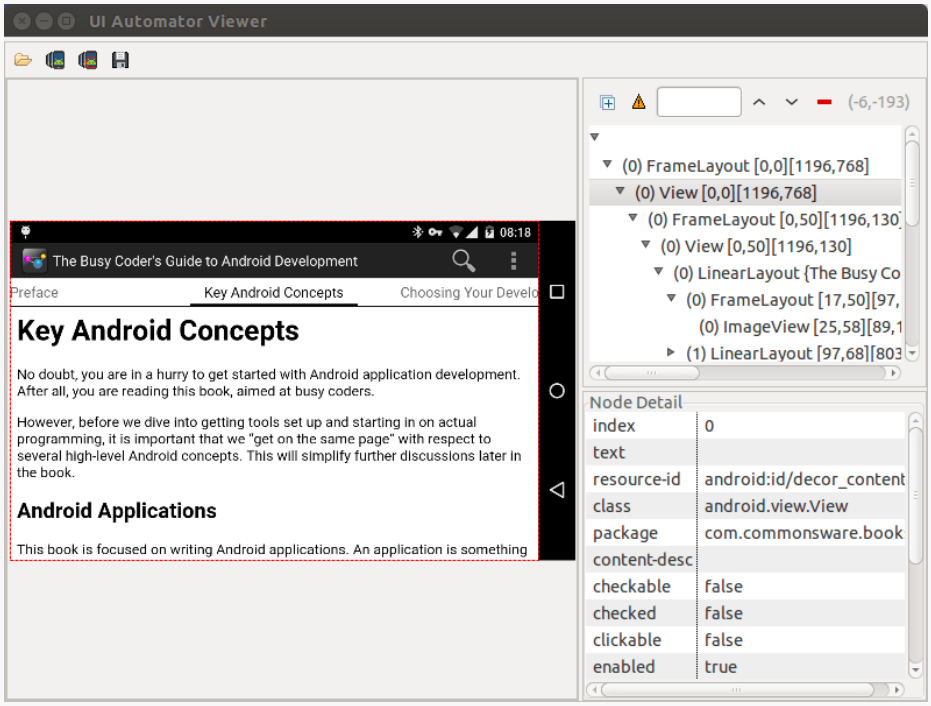


Figure 360: UI Automator Viewer, Showing View Hierarchy of This Book's Reader App

Clicking either on the preview or on the view hierarchy will change the selected view, which shows up with a red dashed outline on the preview. The properties ("Node Detail") pane will then update to show the properties of whatever is newly selected.

This is not only useful for identifying widgets for testing with UI Automator, but it can also be used to determine how some other developer pulled off some interesting UI approach. While simply examining a widget hierarchy is not going to uncover all the other developer's secrets, simply knowing what widgets were used, and some basic properties of those widgets, may give you some ideas for avenues of research.

Measuring Test Coverage

Test coverage is our way of determining whether or not we have adequately tested our code. Part of the work that has gone into the Android Gradle Plugin has been to make obtaining test coverage reports fairly easy, so ideally it is something that you can incorporate into your regular testing regimen.

In this chapter, we will explore the concept of test coverage in general, along with how to generate coverage reports for your Android instrumentation tests.

Prerequisites

Understanding this chapter requires that you have read the chapter on instrumentation testing with JUnit.

Who Tests the Testers?

We use tests to determine if our code works. More generally, we use tests as a way of quantifying the quality of our code. Code that fails the tests is of lower quality than is code that does not fail the tests.

Right?

Suppose we have some Java code that will result in divide-by-zero exception. We have two developers test that code. One developer writes tests and uncovers the exception. The other developer writes tests that bypasses the flawed code, and therefore does not uncover the exception. Here, the code quality is the same, but the *test quality* differs.

If the way we measure code quality is "does it pass the tests", measuring test coverage asks "do the tests adequately test the code?". In the preceding example, either:

- The second developer would have worse test coverage than the first developer, as the second developer clearly is not testing everything, or
- *Both* developers would have poor test coverage, and it just so happens that one stumbled upon the bug ("Even a blind squirrel finds a nut once in a while")

What you want is to have a test suite that has 100% practical coverage. The "practical" qualifier is because there are certain portions of our code that may be impractical to test, because they depend upon certain environmental factors that are different to arrange to happen on demand (e.g., `OutOfMemoryError`). There, the objective is to have as little code specifically dependent upon those factors, moving more of it into code that we can test without requiring those conditions.

Some Types of Test Coverage

Some developers that start in on test coverage think that test coverage is fairly simple to measure: did we run everything? The problem is that "run" has different meanings in different circumstances, and as a result measuring coverage can be done in different ways.

Statement Coverage

The basic approach to measuring test coverage is: did we execute every line of Java code? Clearly, if we never executed a line of code, we did not test that line and have no idea if that line works or not.

Branch Coverage

However, just because we execute a line does not mean that we have executed it under all conditions. Imagine a Java method like this:

```
void doSomething(boolean flag) {
  if (flag) {
    // do one thing
  }
  else {
    // do something else
```

```
  }
}
```

If this method were in our own Java code, we could determine whether we have tested both `true` and `false` cases by means of statement coverage. Either we executed the statements in both branches, or we did not.

However, suppose the method instead looked like this:

```
void doSomething(boolean flag) {
  if (flag) {
    // do one thing
  }

  // do something regardless of the flag value
}
```

Now, 100% statement coverage tells us that we executed the contents of the `if` block. However, it does *not* tell us if we have tested the case where `flag` is `false`, since no additional statements are executed for that case.

Branch coverage, therefore, measures whether our `if` and `switch` statements have covered all scenarios. We might have 100% statement coverage but below 100% branch coverage.

Loop Coverage

The coverage capability integrated into Android Studio offers statement and branch coverage. This does not mean that it includes *all* forms of coverage measurement.

Another common one is loop coverage. Imagine your typical Java `for` loop:

```
void doSomething(int count) {
  for (int i=0;i<count;i++) {
    // do something
  }
}
```

Statement coverage helps here, but only a little. 100% statement coverage would tell us that we executed the code inside the `for` loop. However, like branch coverage, it does not tell us if we are correctly handling the case where `count` is `0`, because we are not executing additional statements in that case.

Also, there are certain types of bugs that only show up if you execute the code inside the loop multiple times. Statement coverage cannot tell us if we tried `count` of `1` and a count greater than `1`, and so there may be some missed bugs.

Loop coverage is usually interpreted as testing three cases:

- Zero passes through the loop
- One pass through the loop
- More than one pass through the loop

Usually, the coverage tool does not know what the logical bounds are for the loop. In theory, there could be a bug that is only found if you execute the loop 349,320 times. However, finding that would require us to try testing every possible `count` value, and executing a couple of billion tests might take a while.

Coverage and Your Instrumentation Tests

The Android Gradle Plugin integrates support for Jacoco, a popular test coverage analyzer tool. Enabling it and measuring your test coverage is very simple.

First, make sure that you are using `0.5` or higher of the `com.android.support.test:rules` artifact, as coverage support was unavailable in some earlier versions.

Then, for the `debug` build type (and others if desired), enable test coverage via the `testCoverageEnabled = true` statement.

The `Testing/Coverage` sample project is a clone of the `Testing/JUnit4` project, with test coverage enabled in the `app` module's `build.gradle` file:

```
apply plugin: 'com.android.application'

dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    androidTestImplementation 'com.android.support.test:rules:1.0.1'
}

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        testApplicationId "com.commonsware.android.gradle.hello.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
```

```
    }

    buildTypes {
        debug {
            testCoverageEnabled = true
        }
    }
}
```

(from Testing/Coverage/app/build.gradle)

At least through Android Studio 2.1, you cannot generate a test coverage report through some simple toolbar button. You need to execute the `createDebugCoverageReport` Gradle task. You can do that from the command line, or you can do that from the Gradle tool, docked by default on the right side of Android Studio:

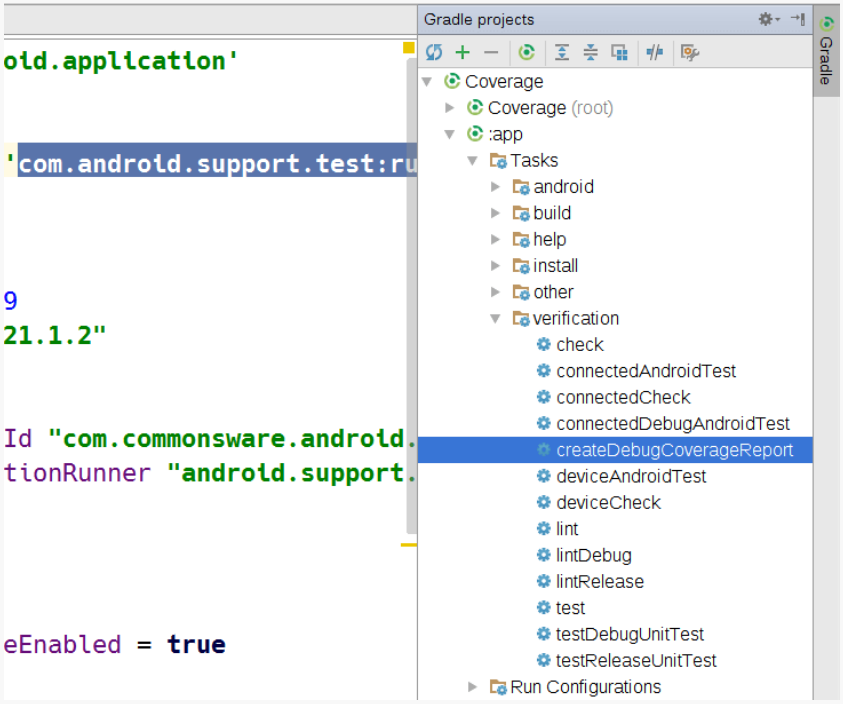


Figure 361: Android Studio Gradle Tool, Showing createDebugCoverageReport Task

Double-clicking a task in the Gradle tool executes that task, just as if you had run it from the command line. The `createDebugCoverageReport` runs your tests and builds a report based on the coverage logging that the build tools add when you have `testCoverageEnabled = true`.

Then, you can browse to the `build/reports/coverage/debug/` directory in your module (replacing `debug` with your build type, if you are using a different build type). That directory holds the HTML report generated by Jacoco of how your test coverage is:

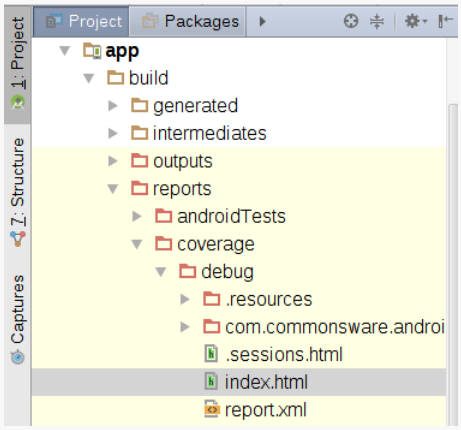


Figure 362: Android Studio, Showing Coverage Report Directory

By default, if you double-click on `index.html`, Android Studio will want to show you the HTML source code in an editor, which is less than useful here. Instead, right-click on it and choose "Open in Browser" to open it in your preferred Web browser:

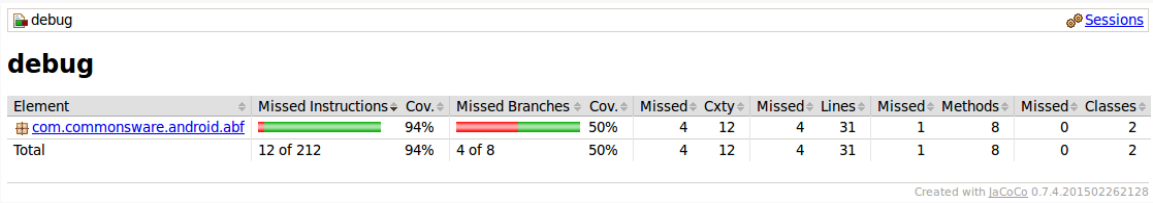


Figure 363: Jacoco Coverage Report, Top Level

Here, we see that we tested one Java package (`com.commonsware.android.abf`) and achieved 94% statement ("instruction") coverage and 50% branch coverage.

You can drill down into the report by clicking on a package to see how individual classes fared:

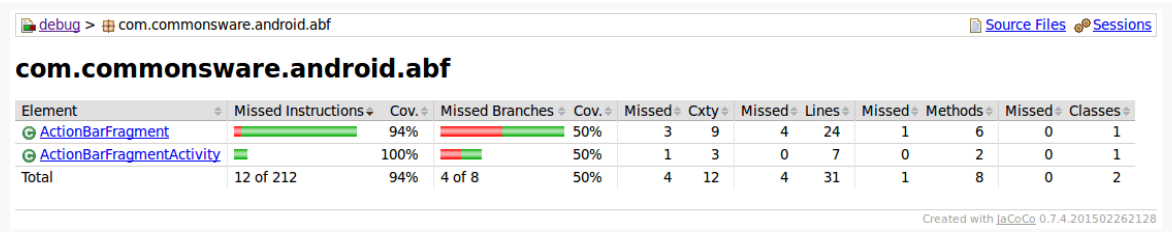


Figure 364: Jacoco Coverage Report, Package Level

Similarly, clicking on a class gives you details of individual methods and constructors:

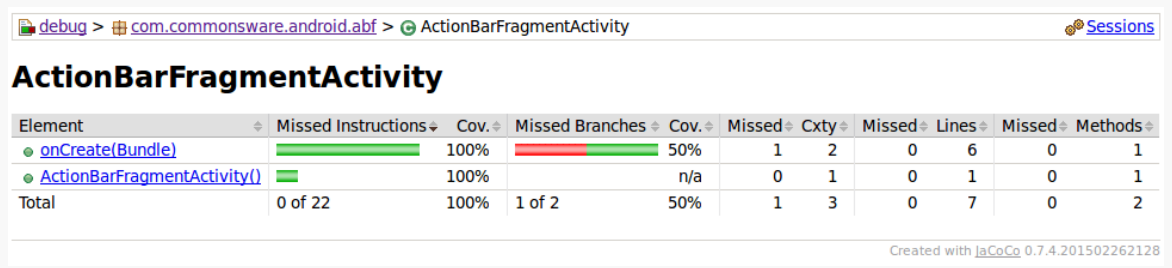


Figure 365: Jacoco Coverage Report, Class Level

Clicking on individual methods will show you highlighted source code of what was covered and what was missed:

Figure 366: Jacoco Coverage Report, Source Highlights

Here, the yellow line with the diamond marker indicates that we did not achieve full branch coverage. In this case, we have not handled a configuration change in the tests, so our fragment always needs to be created.

Unit Testing

Instrumentation testing — basically, all the approaches on testing covered so far in this book — is wonderful, but it has one key problem: it is **slow**.

The problem stems largely from the fact that we are running our Android tests in Android. Android emulators are not speedy, and neither are devices, compared with developer machines, continuous integration servers, and the like.

Unit testing, in Android terms, is taking a subset of our tests out of Android and onto our development machine OS itself, running them just as we would plain Java tests in non-Android Java development. On the plus side, we now have much more machine power to run our tests, including the possibility of running tests in parallel across multiple CPU cores. However, whatever we are running for our development OS probably is not Android, and so attempts to use Android from our tests are doomed to failure... unless we mock Android.

In this chapter, we will explore more of why we might want to set up unit tests, the basics of setting up unit testing for plain old Java objects (POJOs), and how to use Mockito and Robolectric to mock certain things, notably Android itself.

Prerequisites

This chapter assumes that you have read the preceding chapters on testing, particularly the one covering JUnit4.

Also, the examples in this chapter are based on the Retrofit example from the chapter on Internet access, so if you skipped the Retrofit material, you may wish to go back and review that section.

I Thought We Were Already Unit Testing?

If you used the testing techniques outlined previously in this chapter, you used JUnit, probably JUnit4. Given that JUnit has "unit" in the name, you might have thought that you were doing unit testing.

From the standpoint of standard testing terminology, you may very well have been doing unit testing.

However, in Android development, the phrase "unit testing" is reserved for outside-of-Android tests, running on a JVM on your development machine. Tests using JUnit that are run on Android itself are either "instrumentation tests" (if you are working with activities and related components directly) or "integration tests" (if you are using UIAutomator to exercise your app and other apps to see if they interoperate correctly).

Scenario: Clean Architecture

Robert Martin (a.k.a., "Uncle Bob"), in 2012, wrote a seminal blog post outlining what he refers to as "Clean Architecture". Others have posted their own variations on his original theme.

Usually, Clean Architecture is depicted as a set of concentric circles:

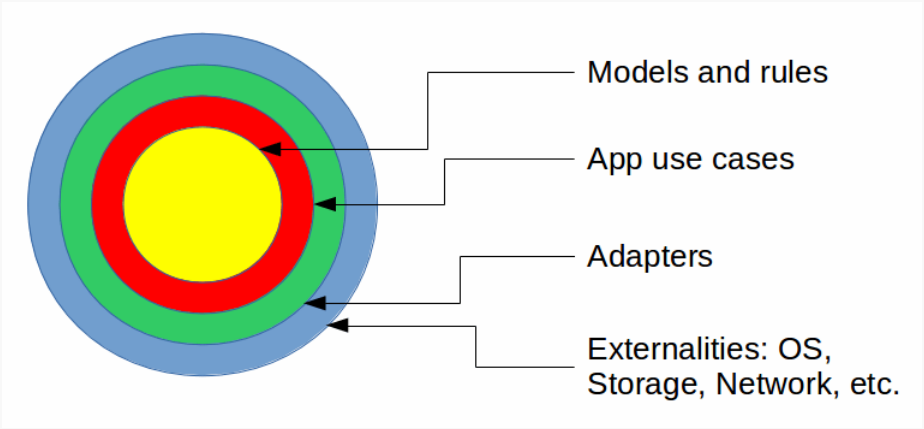


Figure 367: Clean Architecture

The basic rules are:

- The further in from the outside edge you go, the higher level the software becomes
- Nothing from an inner circle can depend upon something in an outer circle

The net of this, from an Android-and-testing standpoint, is that Clean Architecture results in a core set of Java objects that know nothing about Android. Typically, these are model objects and business rules (implemented as some mix of model object methods and separate rule objects). These core objects should know nothing about:

- Activity lifecycles
- Fragments or other UI implementation patterns
- Views or other forms of UI (e.g., Web content in a `WebView`)
- SQLite or any other particular storage engine
- Threads, including idiosyncrasies around the main application thread
- and so on

Even developers who do not adhere religiously to Clean Architecture often strive to have some core objects be "clean enough" and independent from anything specific to Android.

Since these objects have nothing specifically to do with Android, they should be testable outside of Android, even if in production you only intend to use them in Android apps. After all, if they do not need Android, why do you need the testing overhead of Android?

This is where unit testing comes into play, allowing you to set up a set of tests that do not require Android, because the objects being tested to not require Android.

Of course, this is not the only scenario where unit testing can be used. Many apps have some amount of utility code that is not tied too closely to Android. And, as we will see, even some code that has light ties to Android might be able to be tested through unit testing, courtesy of mocking frameworks.

Setting Up Unit Testing

There are three main steps for setting up unit testing: adding JUnit, toggling Android Studio to unit test mode, and creating the `test/` source set for holding your unit tests themselves.

The files shown in this section come from the UnitTest/POJO sample application. This is a clone of the HTTP/Retrofit example from the chapter on Internet access, where we retrieve the latest android-tagged Stack Overflow questions and display them in a ListView.

Adding the Test JUnit Dependency

Gradle, along with the Android Gradle Plugin, are set up to handle unit tests "out of the box". However, they still require you to provide a dependency on JUnit. Partially, that is because you may care about the specific version of JUnit that you use. Partially, that opens the door for possibly using other test engines beyond JUnit.

In previous chapters, we saw that instrumentation tests were handled using an androidTest/ source set and androidTestImplementation dependency statements. Similarly, unit testing is handled via a test/ source set and testImplementation dependency statements. So, to set up JUnit for unit testing, all you need is this in your module's build.gradle file:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        testApplicationId "com.commonsware.android.unittest.tests"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}

dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    implementation 'com.squareup.retrofit:retrofit:1.9.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:rules:1.0.1'
}
```

(from UnitTest/POJO/app/build.gradle)

The version of JUnit will change over time, so the latest-and-greatest one may be newer than what you see above. Also, you may wind up with other testImplementation dependencies — we will see examples of this coming up later in this chapter.

Also note that new Android Studio projects, created through the new-project wizard, may already have this dependency.

You will also notice that this `build.gradle` file is set up to use instrumentation testing as well, with `androidTestImplementation`, `testApplicationId`, and so forth. That is not required for unit testing, but this project uses both testing approaches, to help compare and contrast the results.

Creating the Test Sourceset

New Android Studio projects, created through the new-project wizard, automatically get a `test/` source set added alongside `main/` and `androidTest/`. That `test/` source set will already have an `ExampleUnitTest` class added to the Java package associated with the app.

If you have an existing Android Studio project, though, you will need to set up the `test/` directory, the `test/java/` directory, and add a Java package to that `test/java/` directory yourself. Note that this will be easier if you switch Android Studio to unit test mode. While Android Studio is in instrumentation test mode, it will not recognize a `test/java/` directory as being a viable location for Java source code. That means you will not have a context menu option to add a Java package, forcing you to create each directory yourself, one at a time (e.g., `test/java/com/`, then `test/java/com/commonsware/`, then...). With Android Studio in unit test mode, `test/java/` will be a normal Java source directory, and you can add new Java packages to it from the context menu (right-click over `java/`) as you would in your `main` source set.

Either way, in the end, you should have a `test/` source set as a peer of your `main/` and `androidTest/` source sets:

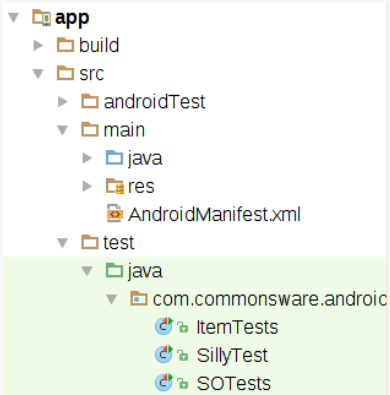


Figure 368: Project View, Showing main/, androidTest/, and test/ Sourcesets

However, only the `main/` source set and your chosen test artifact's source set will show up with normal-looking Java directories. So, for example, with unit testing selected as the test artifact, the source code in `androidTest/` is not really recognized as being Java code that is part of the project:

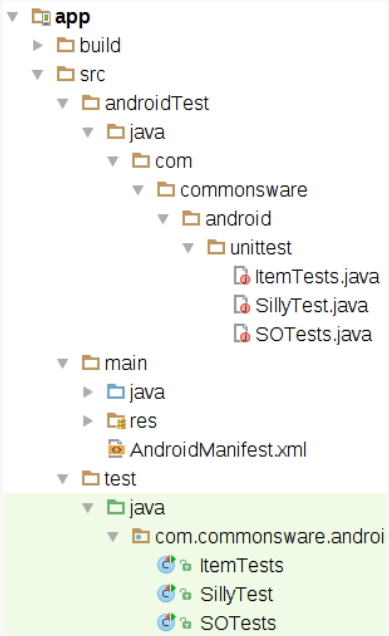


Figure 369: Project View, Showing Unrecognized androidTest/ Java Code

Writing POJO Unit Tests

That setup is all that you need in order to start writing unit tests for your plain old Java objects (POJOs), utility code, or other things that do not depend on Android.

Adding the Test Package

If you had to create the `test/` source set, you will also need to create the `test/java/` directory, and in there create a Java package for your test code.

In Android Studio, this will be much simpler if Android Studio is in unit test mode (versus instrumentation test mode), as then you will have a New > Package option in the context menu, if you right-click over the `java/` directory.

As with other JUnit testing, your choice of package dictates what you can and cannot access of classes and objects being tested:

Test Package	Public	Package-Private	Private
same as class being tested	yes	yes	no
different than class being tested	yes	no	no

In the sample project, the test code is in the same Java package as is the `main` application code, so tests can access `public` and package-private fields, methods, and the like.

Writing a Test Case

We can have a `SillyTest` test case, just as in the chapter on JUnit4. However, we do not need the `@RunWith` annotation on the class:

```
package com.commonsware.android.unittest;

import junit.framework.Assert;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class SillyTest {
  @BeforeClass
  static public void doThisFirstOnlyOnce() {
    // do initialization here, run once for all SillyTest tests
  }

  @Before
  public void doThisFirst() {
    // do initialization here, run on every test method
  }

  @After
  public void doThisLast() {
    // do termination here, run on every test method
  }

  @AfterClass
  static public void doThisLastOnlyOnce() {
    // do termination here, run once for all SillyTest tests
  }

  @Test
  public void thisIsReallySilly() {
```

```
    Assert.assertEquals("bit got flipped by cosmic rays", 1, 1);
  }
}
```

(from UnitTest/POJO/app/src/test/java/com/commonsware/android/unittest/SillyTest.java)

If you created your project through the Android Studio new-project wizard, and it already had a `test/` source set for you, it would have created a similarly-silly `ExampleUnitTest` test case for you:

```
public class ExampleUnitTest {
  @Test
  public void addition_isCorrect() throws Exception {
    assertEquals(4, 2+2);
  }
}
```

Of course, you can start writing your own test cases that are somewhat less silly. Here, we have a test case that confirms the `toString()` behavior of the `Item` class:

```
package com.commonsware.android.unittest;

import junit.framework.Assert;
import org.junit.Test;

public class ItemTests {
  private static final String TITLE="this is a title";
  private static final String URL="https://commonsware.com";

  @Test
  public void iCanHazString() {
    Item item=new Item();

    item.title=TITLE;
    item.link=URL;

    Assert.assertEquals(TITLE, item.toString());
  }
}
```

(from UnitTest/POJO/app/src/test/java/com/commonsware/android/unittest/ItemTests.java)

Testing loading the questions gets a bit tricky, as our `StackOverflowInterface` is set up for asynchronous operation. When we call `questions()` to get the questions, we get control back immediately, and we need to wait for the background thread to

deliver our results. There are a few patterns for handling this. This particular test case uses a `CountDownLatch`:

```
package com.commonsware.android.unittest;

import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
import java.util.concurrent.CountDownLatch;
import retrofit.Callback;
import retrofit.RestAdapter;
import retrofit.RetrofitError;
import retrofit.client.Response;

public class SOTests {
  private CountDownLatch responseLatch;
  private SOQuestions questions;

  @Before
  public void setUp() {
    responseLatch=new CountDownLatch(1);
  }

  @Test(timeout=30000)
  public void fetchQuestions() throws InterruptedException {
    RestAdapter restAdapter=
      new RestAdapter.Builder()
        .setEndpoint("https://api.stackexchange.com")
        .build();
    StackOverflowInterface so=
      restAdapter.create(StackOverflowInterface.class);

    so.questions("android", new Callback<SOQuestions>() {
      @Override
      public void success(SOQuestions soQuestions,
                          Response response) {
        questions=soQuestions;
        responseLatch.countDown();
      }

      @Override
      public void failure(RetrofitError error) {
        responseLatch.countDown();
      }
    });

    responseLatch.await();
```

```
    Assert.assertNotNull(questions);
    Assert.assertEquals(30, questions.items.size());

    for (Item item : questions.items) {
      Assert.assertNotNull(item.title);
      Assert.assertNotNull(item.link);
    }
  }
}
```

(from UnitTest/POJO/app/src/test/java/com/commonsware/android/unittest/SOTests.java)

When the test is `setUp()`, we initialize the `CountDownLatch`, to require one `countDown()` call before the latch is considered to be released.

In our `fetchQuestions()` test method, we go through the same sort of code that `QuestionsFragment` does, creating our `RestAdapter` and `StackOverflowInterface`. When we call `questions()`, we supply an anonymous inner class instance of the `Callback`. In both `success()` and `failure()`, we `countDown()` our `CountDownLatch`. If the call succeeded, we also hold onto the `SOQuestions` model object.

Immediately after calling `fetchQuestions()`, we `await()` on the `CountDownLatch`. The `Callback` will be called on a background thread, so the `await()` call means that we are blocking until such time as we are called with `success()` or `failure()`. Also, as a fail-safe measure, the `@Test` annotation for this test method is configured as `@Test(timeout=30000)`, meaning that if we do not get a response in 30 seconds, we fail the test.

Once we get control after the `success()` or `failure()` call, we confirm and see if we got our 30 questions and that each `Item` seems to be filled out.

Running Unit Tests

Once you have one or more unit tests, you can start thinking about running them and seeing if they work. Running unit tests does not require a device or emulator, as these tests are running on your development machine's OS directly (in a standard Java VM), not on Android.

From Android Studio

From the project tree, right-clicking over a class or a package will give you a context menu option to run the tests in that class or package:

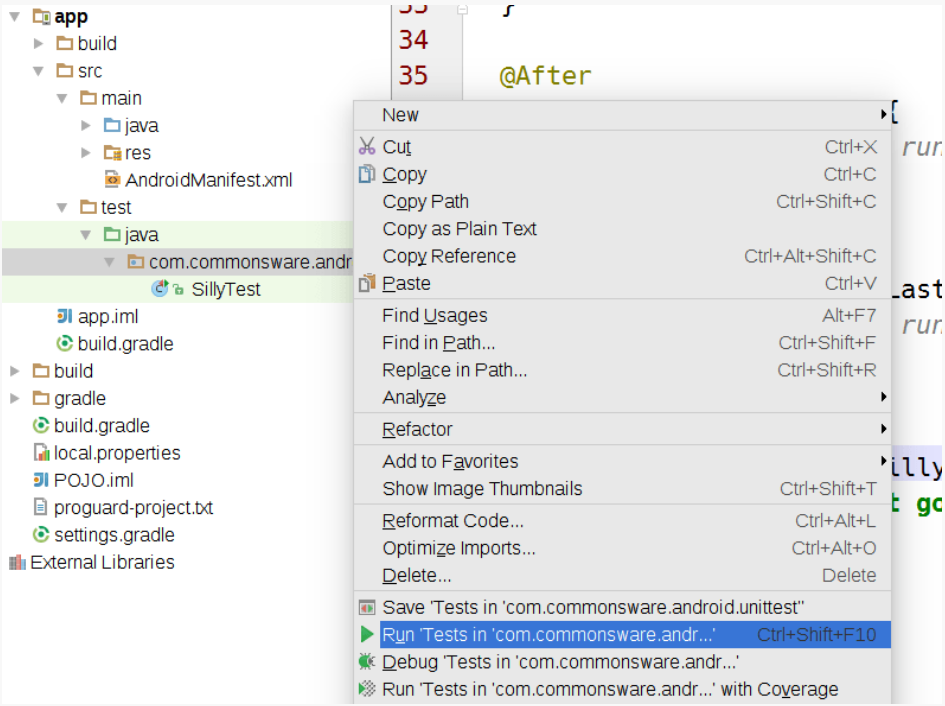


Figure 370: Run Tests Context Menu Item for Test Package

You can even right-click over the name of a method in your test class and have an option for running just that method:

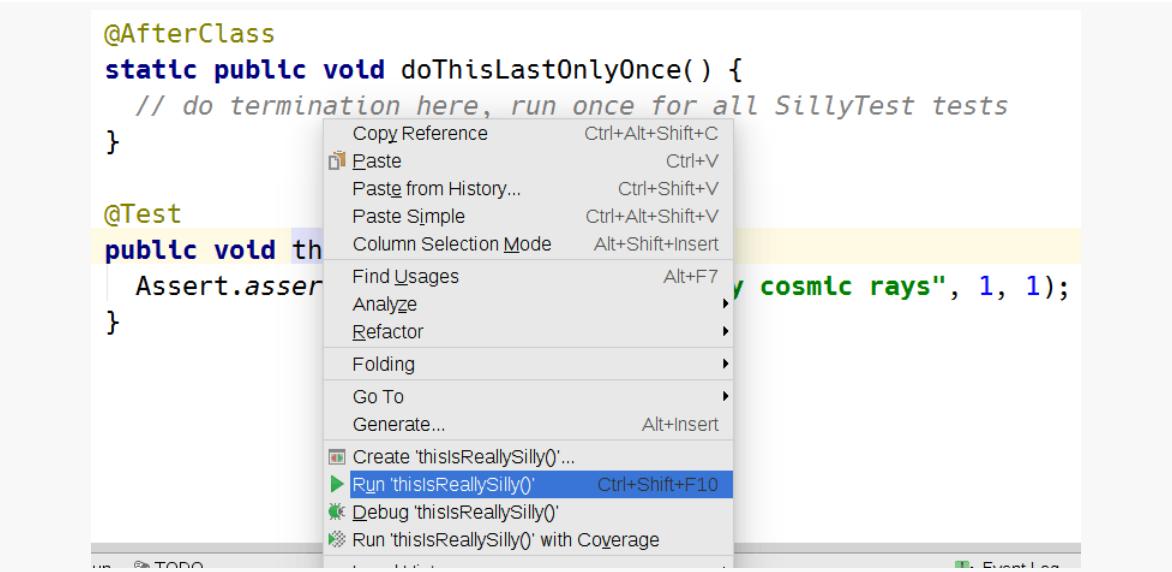


Figure 371: Run Test Context Menu Item for Test Method

While convenient, this will clutter up your run configurations drop-down:

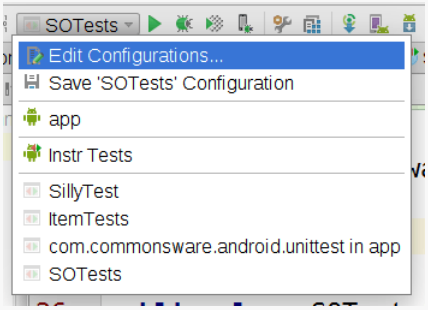


Figure 372: Run Configurations, After Running Unit Tests

The entries with the faded-out icons represent run configurations that were added dynamically based on your right-click test runs. The most recent of those becomes the current run configuration (shown here as "SOTests"), and you have an option to "save" that and make it a regular run configuration.

If you try running the various unit tests for the sample app, `SillyTest` and `ItemTests` work, but `SOTests` does not. Apparently, Retrofit depends too much on Android, since the same test code succeeds when run as an instrumentation test. We will discuss how to deal with that problem later in this chapter.

From the Command Line

The `test` task in Gradle runs all of your unit tests for the module in which you run that task. The command line output will show you a summary of the results, in this case demonstrating the failure of `SOTests`:

```
$ gradle test
:app:preBuild UP-TO-DATE
:app:preDebugBuild UP-TO-DATE
:app:checkDebugManifest
:app:prepareDebugDependencies
:app:compileDebugAidl
:app:compileDebugRenderscript
:app:generateDebugBuildConfig
:app:generateDebugAssets UP-TO-DATE
:app:mergeDebugAssets
:app:generateDebugResValues
:app:generateDebugResources
:app:mergeDebugResources
:app:processDebugManifest
:app:processDebugResources
:app:generateDebugSources
:app:compileDebugJavaWithJavac
:app:preDebugUnitTestBuild UP-TO-DATE
:app:prepareDebugUnitTestDependencies
:app:compileDebugUnitTestJavaWithJavac
:app:processDebugJavaRes UP-TO-DATE
:app:processDebugUnitTestJavaRes UP-TO-DATE
:app:compileDebugUnitTestSources
:app:mockableAndroidJar
:app:assembleDebugUnitTest
:app:testDebugUnitTest

com.commonsware.android.unittest.SOTests > fetchQuestions FAILED
    junit.framework.AssertionFailedError at SOTests.java:60

3 tests completed, 1 failed
:app:testDebugUnitTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:testDebugUnitTest'.
> There were failing tests. See the report at: file:///home/mmurphy/stuff/CommonsWare/
books/Omnibus/samples/UnitTest/POJO/app/build/reports/tests/debug/index.html

* Try:
```

```
Run with --stacktrace option to get the stack trace. Run with --info or --debug
option to get more log output.

BUILD FAILED

Total time: 14.188 secs
```

We see that three tests completed, but one (`SOTests`) failed.

In `build/output/reports/tests/` will be HTML reports showing the results of the tests:

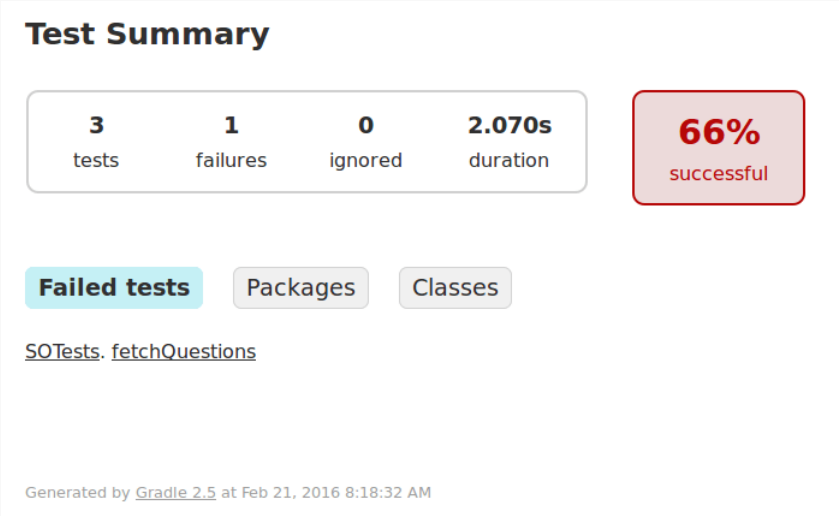


Figure 373: Unit Test Report, Summary

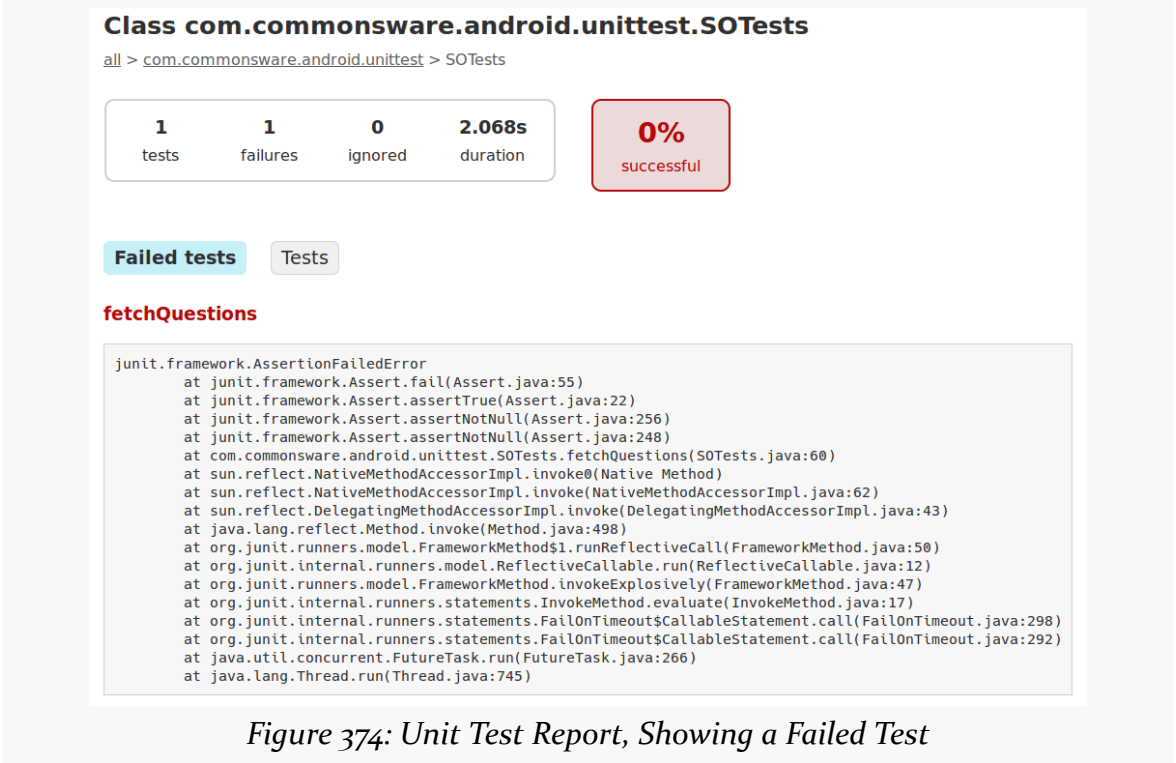


Figure 374: Unit Test Report, Showing a Failed Test

There are XML files in `build/output/test-results/` that contain the same basic information. These are mostly designed for use by tools, such as perhaps a CI server.

Mocking Android

Unit tests that go beyond stuff in common between the JVM and Android are going to have problems, such as the Retrofit example described above. For pure POJOs, this will not be a major limitation. But you might have other code that has little real connection to Android that you would like to test using unit testing, for the faster speed. However, unit testing is fairly unforgiving: "little real connection" is not "no connection", and so your tests will fail.

Hence, to fix unit testing, we need to mock Android.

Why Are We Being Mean to Android?

In this case, "mock" is not a synonym for "taunt".

Instead, "mock" refers to creating mock objects. Wikipedia describes this as:

> In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.

(from a February 2016 edition of the page)

For example, in many places in Android, we need a `Context`. Sometimes, we do not really use the `Context` at all ourselves — it is input to some lower layer of code, and therefore we accept it as input to our layer and pass it along. In those cases, perhaps a mock `Context` will suffice to allow our tests to run. Or, perhaps endowing the mock `Context` with some limited amount of test-defined functionality will suffice to allow our tests to run.

There are many mock frameworks for programming environments, including a few for Java, and some of those Java ones are Android-friendly. This section will look at one of those, Mockito. This section will also look at Robolectric, a framework specifically for mocking the Android SDK.

Mockito

Mockito is a general-purpose mocking library for Java that is officially supported by the Android tools team for use with Android unit testing. While there are other mocking libraries for Java (e.g., jMock, EasyMock), you may wish to start with Mockito given its official support status.

Why Mockito?

The idea behind any Java mocking library is to be able to create objects that, from a compilation standpoint, behave as do the real objects, but have test-controlled responses to methods (a.k.a., "stubs").

The quintessential Java example is mocking a `List`:

```
List fakeList=mock(List.class);

when(fakeList.get(0)).thenReturn(1337);
```

Here, we use an imported `static` `mock()` method to create a mock implementation of the `List` interface. `when()` captures a particular invocation that we wish to stub

out (in this case, getting the oth item in the list), and `thenReturn()` indicates what the return value should be for that invocation (in this case, `1337`). Later on, we can test our behavior:

```
Assert.assertEquals(1337, fakeList.get(0)); // succeeds
Assert.assertEquals(1337, fakeList.get(1)); // fails, as get(1) returns null
```

Since we taught the mock how to respond to `get(0)`, it returns `1337`. Anything else we try doing with the mock will result in some default behavior; in this case, calling `get()` for any other index will return `null`, since we have not defined values for any other indexes.

For limited tests like this, we are not really testing much in the way of actual app functionality. If anything, we are testing that Mockito is capable of mocking things. However, suppose instead that we did this:

```
OurClass sumthin=new OurClass();

Assert.assertEquals(1787569, sumthin.squareTheFirst(fakeList));
```

Here, we have a `squareTheFirst()` method implemented on some class of ours `OurClass`. As it turns out, the implementation of `squareTheFirst()` is to grab the oth element out of the supplied `List` and return the square of that integer value. Now we are testing actual application logic, confirming that our square is being computed properly.

Of course, in this case, it would be just as easy to create an `ArrayList`, rather than mess with a mock. However, there are plenty of cases where it would be too much work to create an instance of the class, including cases where it is nearly impossible. For example, we cannot create our own instances of system services, like `AlarmManager` or `NotificationManager`. If we want to test code that works with those, we are far better served using a mocking library like Mockito.

Setting Up Mockito

The `UnitTest/Mockito` sample application is based on the `POJO` one from earlier in the chapter. However, this version adds Mockito, specifically to help us play around with a mock version of Retrofit.

The Android tools team seems to be endorsing Mockito 1.x, as at the present time (February 2016), Mockito 2.0 is still in beta. Adding Mockito, therefore, is a matter of

adding a `testImplementation` statement to pull in an appropriate version of `mockito-core`:

```
dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    implementation 'com.squareup.retrofit:retrofit:1.9.0'
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.mockito:mockito-core:2.10.0'
    androidTestImplementation 'com.android.support.test:rules:1.0.1'
}
```

(from UnitTest/Mockito/app/build.gradle)

Using Mockito in Unit Tests

Let's now use Mockito to create an `SOTests` that works, albeit using a fake Retrofit.

To use Mockito in a JUnit4 test class, you need to add the `@RunWith(MockitoJUnitRunner.class)` annotation to the class, to have the class run using a dedicated JUnit4 test runner that is Mockito-enabled. So, `SOTests` needs that annotation:

```
@RunWith(MockitoJUnitRunner.class)
public class SOTests {
```

(from UnitTest/Mockito/app/src/test/java/com/commonsware/android/unittest/SOTests.java)

We want a mock `StackOverflowInterface` object that we can use to call our `questions()` method and retrieve mock questions. There are two main ways in Mockito to create mock objects:

- the `mock()` method cited earlier
- the `@Mock` annotation

`SOTests` applies the latter, so one of the fields in the test class is a `@Mock` of `StackOverflowInterface`:

```
@RunWith(MockitoJUnitRunner.class)
public class SOTests {
  private CountDownLatch responseLatch;
  private SOQuestions questions;
  @Mock StackOverflowInterface mockSO;
```

(from UnitTest/Mockito/app/src/test/java/com/commonsware/android/unittest/SOTests.java)

We then need to teach our mock `StackOverflowInterface` how to return questions as needed.

If we were using Retrofit's synchronous API, the `questions()` method on `StackOverflowInterface` would return the `SOQuestions` object representing the results of our REST API call. In that case, mocking `StackOverflowInterface` could be (comparatively) simple, something like:

```
SOQuestions fakeQuestions=new SOQuestions();

fakeQuestions.items=new ArrayList<Item>();

Item fakeItem=new Item();

fakeItem.link="https://commonsware.com";
fakeItem.title="How Do I Fake It to Make It?";
fakeQuestions.items.add(fakeItem);

when(mockSO.questions()).thenReturn(fakeQuestions);
```

Here, we build up an `SOQuestions` instance containing a single `Item`, and we teach the `mockSO` object to return that in response to a call to `questions()`.

However, we are using Retrofit's asynchronous API, where we supply a `Callback` as a parameter. The `questions()` method is declared as `void`, so it does not return a response, and we will eventually be called with `success()` on our `Callback`.

We can mock that with Mockito, but it is more complicated:

```
    doAnswer(new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation)
        throws Throwable {
        SOQuestions fakeQuestions=new SOQuestions();

        fakeQuestions.items=new ArrayList<Item>();

        Item fakeItem=new Item();

        fakeItem.link="https://commonsware.com";
        fakeItem.title="How Do I Fake It to Make It?";
        fakeQuestions.items.add(fakeItem);

        Callback<SOQuestions> realCB=
          (Callback<SOQuestions>)invocation.getArguments()[1];
```

```
      realCB.success(fakeQuestions, null);

      return(null);
    }
  }).when(mockSO).questions(eq("android"), any(Callback.class));
```

(from UnitTest/Mockito/app/src/test/java/com/commonsware/android/unittest/SOTests.java)

Let's trim this back to the essence of what we are doing:

```
doAnswer(...).when(mockSO).questions(...)
```

The `doAnswer()` flow is a way of handling `void` methods, as `thenReturn()` does not work, since a `void` method does not return anything. `doAnswer()` is where we do the "work" that the mock needs to do, in this case to call our `Callback`, as we will see shortly.

Chaining `questions()` onto the result of `when()` is another way of indicating a call that we are stubbing. However, in this case, our parameters are not simple primitives, like the `0` in `get(0)` from the `List` example depicted earlier in this section. We can provide information to `questions()` to indicate *which* `questions()` calls will get the answer provided by `doAnswer()`.

```
doAnswer(...).when(mockSO).questions(eq("android"), any(Callback.class))
```

The real `questions()` method takes a `String` representing the Stack Overflow tag of interest, plus the `Callback`. The particular invocation of `questions()` that we are stubbing will be used when `questions()` is called on our mock `StackOverflowInterface`, where the first parameter equals (`eq()`) `"android"`, and where the second parameter is `any()` instance of `Callback.class`. In principle, we could provide other stubs for other tags (e.g., `questions(eq("ios"), any(Callback.class))`), but that is beyond the scope of what we are doing here.

The value passed into `doAnswer()` is an anonymous inner class implementation of `Answer`. That object's `answer()` method will be called when our mock is called with a matching `questions()` call. It is our job, in `answer()`, to do whatever the mock needs to do to satisfy our tests.

Most of what is here builds up the same fake `SOQuestions` object as illustrated earlier. To actually pass that to the `Callback`, though, we need to:

- Get the `Callback` object in question, by calling `getArguments()` on the supplied `InvocationOnMock` object (which collects all the parameters passed into `questions()`) and gets the second entry from that array of objects
- Casts that to the correct type
- Calls `success()`, passing in the fake `SOQuestions`, plus a `null` value for the Retrofit `Response` object, as we are not using that

At that point, we can run our test, adjusting it to expect one `Item` instead of 30 as we were originally expecting:

```
    mockSO.questions("android", new Callback<SOQuestions>() {
      @Override
      public void success(SOQuestions soQuestions,
                          Response response) {
        questions=soQuestions;
        responseLatch.countDown();
      }

      @Override
      public void failure(RetrofitError error) {
        responseLatch.countDown();
      }
    });

    responseLatch.await();

    Assert.assertNotNull(questions);
    Assert.assertEquals(1, questions.items.size());

    for (Item item : questions.items) {
      Assert.assertNotNull(item.title);
      Assert.assertNotNull(item.link);
    }
```

(from UnitTest/Mockito/app/src/test/java/com/commonsware/android/unittest/SOTests.java)

In this case, everything should be happening on the same thread, and so it is likely that the `CountDownLatch` is superfluous. However, it does not cause us any particular harm here, and it keeps the code more closely aligned with the implementation in the instrumentation tests.

The whole test class, therefore, looks like this:

```
package com.commonsware.android.unittest;
```

```
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.runners.MockitoJUnitRunner;
import org.mockito.stubbing.Answer;
import java.util.ArrayList;
import java.util.concurrent.CountDownLatch;
import retrofit.Callback;
import retrofit.RetrofitError;
import retrofit.client.Response;
import static org.mockito.Matchers.any;
import static org.mockito.Matchers.eq;
import static org.mockito.Mockito.doAnswer;

@RunWith(MockitoJUnitRunner.class)
public class SOTests {
  private CountDownLatch responseLatch;
  private SOQuestions questions;
  @Mock StackOverflowInterface mockSO;

  @Before
  public void setUp() {
    responseLatch=new CountDownLatch(1);
  }

  @Test(timeout=30000)
  public void fetchQuestions() throws InterruptedException {
    doAnswer(new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation)
        throws Throwable {
        SOQuestions fakeQuestions=new SOQuestions();

        fakeQuestions.items=new ArrayList<Item>();

        Item fakeItem=new Item();

        fakeItem.link="https://commonsware.com";
        fakeItem.title="How Do I Fake It to Make It?";
        fakeQuestions.items.add(fakeItem);

        Callback<SOQuestions> realCB=
          (Callback<SOQuestions>)invocation.getArguments()[1];

        realCB.success(fakeQuestions, null);
```

```
      return(null);
    }
  }).when(mockSO).questions(eq("android"), any(Callback.class));

  mockSO.questions("android", new Callback<SOQuestions>() {
    @Override
    public void success(SOQuestions soQuestions,
                        Response response) {
      questions=soQuestions;
      responseLatch.countDown();
    }

    @Override
    public void failure(RetrofitError error) {
      responseLatch.countDown();
    }
  });

  responseLatch.await();

  Assert.assertNotNull(questions);
  Assert.assertEquals(1, questions.items.size());

  for (Item item : questions.items) {
    Assert.assertNotNull(item.title);
    Assert.assertNotNull(item.link);
  }
 }
}
```

(from UnitTest/Mockito/app/src/test/java/com/commonsware/android/unittest/SOTests.java)

If you run the `fetchQuestions()` test, it works. Of course, it is not really testing anything other than Mockito itself. In this particular app, the only thing we are using Retrofit for is to obtain a list of model objects to put in an `ArrayAdapter` to display in a `ListView`. Mockito cannot readily help us test whether our `ArrayAdapter` is inflating layouts properly, or whether `ListView` is using the `ArrayAdapter` properly. At best, we would wind up creating massive mocks that, once again, mostly just have us testing whether our mocks work, not our actual business logic.

This is not to say that Mockito is useless, but rather that its utility is for lightly extending the scope of what we can test of our POJOs in unit testing.

Mockito is a fairly large and complex library. There are multiple books available covering Mockito, should you need more than the project documentation and similar online sources. Just remember that Mockito is for general Java development and is not Android-specific.

Robolectric

Mockito mocks anything you want, so long as you are the one doing the mocking.

Robolectric mocks a part of the Android SDK for you. In particular, it mocks the setup and teardown of activities and services, so you can confirm that they are initialized properly through unit tests, instead of instrumentation tests. Robolectric also supports a limited amount of user input testing — mostly limited to click events — so you can determine whether or not your activity is reacting as expected.

On the one hand, Robolectric lets you set up some Android-specific unit tests "out of the box". On the other hand, the depth and breadth of its mocking is fairly limited, which will steer more of your tests back to instrumentation testing, where you have a full Android SDK at your disposal.

The `UnitTest/Robolectric` sample application is based on the one shown in the chapter on JUnit4, where we have 25 Latin words that we are showing in a list, and we want to test that the activity is coming up as expected.

Setting up Robolectric

You will need to add a `testImplementation` directive to your `build.gradle` file to pull in Robolectric, plus a `testOptions` closure in the `android` closure, advertising that you want resources to be available to you while testing:

```
apply plugin: 'com.android.application'

dependencies {
    implementation "com.android.support:support-fragment:27.0.2"
    testImplementation 'junit:junit:4.12'
    testImplementation "org.robolectric:robolectric:3.7.1"
    androidTestImplementation 'com.android.support.test:rules:1.0.1'
}

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
```

```
        testApplicationId "com.commonsware.android.gradle.hello.robolectric"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }

    testOptions {
        unitTests {
            includeAndroidResources = true
        }
    }
}
```

(from UnitTest/Robolectric/app/build.gradle)

Choosing an API Level

A key decision that you will need to make, before writing any tests, is what API level you want Robolectric to mock. Robolectric does not have mocks for all API levels, as creating their roster of mocks takes work.

Unfortunately, the roster of supported API levels does not appear to be documented.

The Roboletric SdkConfig class class is the closest that we have to documentation, as it has a `static` block that sets up the supported SDKs. The following is from Robolectric 3.7.1:

```
private static final Map<Integer, SdkVersion> SUPPORTED_APIS =
Collections.unmodifiableMap(new HashMap<Integer, SdkVersion>() {
  {
    addSdk(Build.VERSION_CODES.JELLY_BEAN, "4.1.2_r1", "r1", "REL");
    addSdk(Build.VERSION_CODES.JELLY_BEAN_MR1, "4.2.2_r1.2", "r1", "REL");
    addSdk(Build.VERSION_CODES.JELLY_BEAN_MR2, "4.3_r2", "r1", "REL");
    addSdk(Build.VERSION_CODES.KITKAT, "4.4_r1", "r2", "REL");
    addSdk(Build.VERSION_CODES.LOLLIPOP, "5.0.2_r3", "r0", "REL");
    addSdk(Build.VERSION_CODES.LOLLIPOP_MR1, "5.1.1_r9", "r2", "REL");
    addSdk(Build.VERSION_CODES.M, "6.0.1_r3", "r1", "REL");
    addSdk(Build.VERSION_CODES.N, "7.0.0_r1", "r1", "REL");
    addSdk(Build.VERSION_CODES.N_MR1, "7.1.0_r7", "r1", "REL");
    addSdk(Build.VERSION_CODES.O, "8.0.0_r4", "r1", "REL");
    addSdk(Build.VERSION_CODES.O_MR1, "8.1.0", "r4402310", "OMR1");
  }

  private void addSdk(int sdkVersion, String androidVersion, String
frameworkSdkBuildVersion,
      String codeName) {
      put(sdkVersion, new SdkVersion(androidVersion, frameworkSdkBuildVersion,
codeName));
```

```
  }
});
```

This translates to support for API Level 16-19 and 21-27.

You will need to choose a suitable API level for your use. By default, Robolectric will use your `targetSdkVersion`, which may or may not be one of the supported API levels. You will need to know what Robolectric API level to use when you start setting up your tests.

Writing Robolectric Tests

At this point, you can start writing tests that use Robolectric. As with the Mockito tests and other unit tests, these will go in your `test/` source set.

The sample project has a `DemoActivityTest` that, in theory, would mimic the `DemoActivityTest` from the instrumentation tests in the `androidTest/` source set:

```
package com.commonsware.android.abf.test;

import android.widget.ListView;
import com.commonsware.android.abf.ActionBarFragmentActivity;
import com.commonsware.android.abf.BuildConfig;
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.robolectric.Robolectric;
import org.robolectric.RobolectricTestRunner;
import org.robolectric.annotation.Config;
import static android.os.Build.VERSION_CODES.JELLY_BEAN;

@RunWith(RobolectricTestRunner.class)
@Config(sdk=JELLY_BEAN)
public class DemoActivityTest {
  private ListView list=null;

  @Before
  public void setUp() throws Exception {
    ActionBarFragmentActivity activity=
      Robolectric.setupActivity(ActionBarFragmentActivity.class);

    list=activity.findViewById(android.R.id.list);
  }
```

```
  @Test
  public void listCount() {
    Assert.assertEquals(25, list.getAdapter().getCount());
  }
}
```

(from UnitTest/Robolectric/app/src/test/java/com/commonsware/android/abf/test/DemoActivityTest.java)

Your test case classes need two annotations. One is `@RunWith(RobolectricTestRunner.class)`, to tell the unit test system to use Robolectric's test runner rather than the stock JUnit one. If you *also* plan on using Mockito, since you cannot use two test runners, use the Robolectric one and add `MockitoAnnotations.initMocks(this);` to a `@Before` method of your test case, to initialize Mockito.

You can optionally include a `@Config` annotation, to configure Robolectric behavior. The one shown here specifies that we want to test as if we are running on API Level 16 (`sdk=JELLY_BEAN`), instead of our `targetSdkVersion`. The `sdk` property is not required, if your `targetSdkVersion` is a supported value. However, you may wish to still specify it on the tests, so your tests are isolated from changes that you might make to the `targetSdkVersion`.

To get a fully-initialized mock activity, call `Robolectric.setupActivity()`, providing the Java class object for the activity in question (e.g., `ActionBarFragmentActivity.class`). This works similarly to calling `getActivity()` on an `ActivityTestRule` in a JUnit4 instrumentation test: you get an activity, with the appropriate data type, ready for testing. In this case, we retrieve the `ListView` and, in a `@Test` method, ensure that the adapter in the `ListView` has 25 entries.

The instrumentation test edition of `DemoActivityTest` also tests key and touch events. While Robolectric supports `performClick()` calls on views to simulate click events, simulating key and touch events does not appear to be well-supported, which is why the Robolectric test case skips them.

Running Robolectric Tests

You run the Robolectric tests the same way as any other unit tests, such as by right-clicking over the test case class and choosing the run option.

The first time that you run the tests for your module, Robolectric will download a bunch of stuff:

```
/usr/lib/jvm/java-8-oracle/bin/java -ea -Didea.launcher.port=7535
-Didea.launcher.bin.path=/home/mmurphy/android-studio/bin -Dfile.encoding=UTF-8
-classpath /home/mmurphy/android-studio/lib/idea_rt.jar:/home/mmurphy/android-studio/
plugins/junit/lib/junit-rt.jar:/opt/android-sdk-linux/platforms/android-19/data/
res:/home/mmurphy/stuff/CommonsWare/books/Omnibus/samples/UnitTest/Robolectric/app/
build/intermediates/classes/test/debug:/home/mmurphy/stuff/CommonsWare/books/Omnibus/
samples/UnitTest/Robolectric/app/build/intermediates/classes/debug:/home/
mmurphy/.gradle/caches/modules-2/files-2.1/org.apache.maven/maven-ant-tasks/2.1.3/
b09be554228d66d208e5fef5266844aacf443abc/maven-ant-tasks-2.1.3.jar:/home/
mmurphy/.gradle/caches/modules-2/files-2.1/org.apache.ant/ant/1.8.0/
7b456ca6b93900f96e58cc8371f03d90a9c1c8d1/ant-1.8.0.jar:/home/mmurphy/.gradle/caches/
modules-2/files-2.1/com.ibm.icu/icu4j/53.1/786d9055d4ca8c1aab4a7d4ac8283f973fd7e41f/
icu4j-53.1.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/
com.google.android.apps.common.testing.accessibility.framework/
accessibility-test-framework/1.0/28162aae36f8ba5903adadfb570313e1f1be852e/
accessibility-test-framework-1.0.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/
org.robolectric/robolectric-resources/3.0/1ab609054aab67cd13a434567467f4b4774f2465/
robolectric-resources-3.0.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/
org.ow2.asm/asm-commons/5.0.1/7b7147a390a93a14d2edfdcf3f7b0e87a0939c3e/
asm-commons-5.0.1.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/org.apache.ant/
ant-launcher/1.8.0/8b53ba16fa62fb1034da8f1de200ddc407c8381/
ant-launcher-1.8.0.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/
org.robolectric/robolectric-annotations/3.0/2a6cfc072d7680694c1ff893c5dc8fec33163110/
robolectric-annotations-3.0.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/
com.almworks.sqlite4java/sqlite4java/0.282/745a7e2f35fdbe6336922e0d492c979dbbfa74fb/
sqlite4java-0.282.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/org.ow2.asm/
asm-tree/5.0.1/1b1e6e9d869acd704056d0a4223071a511c619e6/asm-tree-5.0.1.jar:/home/
mmurphy/.gradle/caches/modules-2/files-2.1/org.ow2.asm/asm/5.0.1/
2fd56467a018aafe6ec6a73ccba520be4a7e1565/asm-5.0.1.jar:/home/mmurphy/.gradle/caches/
modules-2/files-2.1/org.robolectric/shadows-core/3.0/
9dfa881bfd1796afa28204ef1a5ed7e3de992612/shadows-core-3.0.jar:/home/mmurphy/.gradle/
caches/modules-2/files-2.1/org.ow2.asm/asm-analysis/5.0.1/
e286fbee48efacb4e7c175f7948d9d8b2ab52352/asm-analysis-5.0.1.jar:/home/mmurphy/.gradle/
caches/modules-2/files-2.1/org.robolectric/robolectric-utils/3.0/
4bcecd8115fe7296088bb1636e6cbd7ae8927392/robolectric-utils-3.0.jar:/home/
mmurphy/.gradle/caches/modules-2/files-2.1/com.ximpleware/vtd-xml/2.11/
ee5bcf62c1acf76434ee9f1c67a840bafef72a6d/vtd-xml-2.11.jar:/home/mmurphy/.gradle/
caches/modules-2/files-2.1/org.hamcrest/hamcrest-core/1.3/
42a25dc3219429f0e5d060061f71acb49bf010a0/hamcrest-core-1.3.jar:/home/mmurphy/.gradle/
caches/modules-2/files-2.1/org.ow2.asm/asm-util/5.0.1/
7c8caddfbd0b2d7b844f8fcc75175b9cb9cf4724/asm-util-5.0.1.jar:/home/mmurphy/.gradle/
caches/modules-2/files-2.1/org.bouncycastle/bcprov-jdk16/1.46/
ce091790943599535cbb4de8ede84535b0c1260c/bcprov-jdk16-1.46.jar:/home/mmurphy/.gradle/
caches/modules-2/files-2.1/junit/junit/4.12/2973d150c0dc1fefe998f834810d68f278ea58ec/
junit-4.12.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/org.mockito/
mockito-core/1.10.19/e8546f5bef4e061d8dd73895b4e8f40e3fe6effe/
mockito-core-1.10.19.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/
org.robolectric/robolectric/3.0/f888cea3bc1a24110e315eb9827ab593610ea62f/
```

```
robolectric-3.0.jar:/home/mmurphy/.gradle/caches/modules-2/files-2.1/org.objenesis/
objenesis/2.1/87c0ea803b69252868d09308b4618f766f135a96/objenesis-2.1.jar:/home/
mmurphy/stuff/CommonsWare/books/Omnibus/samples/UnitTest/Robolectric/app/build/
intermediates/sourceFolderJavaResources/test/debug:/home/mmurphy/stuff/CommonsWare/
books/Omnibus/samples/UnitTest/Robolectric/app/build/intermediates/
sourceFolderJavaResources/debug:/home/mmurphy/stuff/CommonsWare/books/Omnibus/samples/
UnitTest/Robolectric/build/generated/mockable-android-19.jar
com.intellij.rt.execution.application.AppMain
com.intellij.rt.execution.junit.JUnitStarter -ideVersion5
com.commonsware.android.abf.test.DemoActivityTest
Downloading: org/robolectric/shadows-core/3.0/shadows-core-3.0.pom from repository
sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 14K from sonatype
Downloading: org/robolectric/robolectric-shadows/3.0/robolectric-shadows-3.0.pom from
repository sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 1K from sonatype
Downloading: org/robolectric/robolectric-parent/3.0/robolectric-parent-3.0.pom from
repository sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 12K from sonatype
Downloading: org/robolectric/robolectric-annotations/3.0/
robolectric-annotations-3.0.pom from repository sonatype at https://oss.sonatype.org/
content/groups/public/
Transferring 1K from sonatype
Downloading: org/robolectric/robolectric-utils/3.0/robolectric-utils-3.0.pom from
repository sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 2K from sonatype
Downloading: org/robolectric/robolectric-resources/3.0/robolectric-resources-3.0.pom
from repository sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 2K from sonatype
Downloading: org/robolectric/shadows-core/3.0/shadows-core-3.0-16.jar from repository
sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 2587K from sonatype
Downloading: org/robolectric/robolectric-annotations/3.0/
robolectric-annotations-3.0.jar from repository sonatype at https://oss.sonatype.org/
content/groups/public/
Transferring 10K from sonatype
Downloading: org/robolectric/robolectric-utils/3.0/robolectric-utils-3.0.jar from
repository sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 40K from sonatype
Downloading: org/robolectric/robolectric-resources/3.0/robolectric-resources-3.0.jar
from repository sonatype at https://oss.sonatype.org/content/groups/public/
Transferring 146K from sonatype

Process finished with exit code 0
```

This material is cached, so future runs will skip it.

If you get a crash akin to this:

```
java.lang.UnsupportedOperationException: Robolectric does not support API level 14.
  at org.robolectric.internal.SdkConfig.<init>(SdkConfig.java:42)
  at org.robolectric.RobolectricTestRunner.runChild(RobolectricTestRunner.java:187)
  at org.robolectric.RobolectricTestRunner.runChild(RobolectricTestRunner.java:54)
  at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
  at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
  at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
  at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
  at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
  at org.robolectric.RobolectricTestRunner$1.evaluate(RobolectricTestRunner.java:152)
  at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
  at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
  at
com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:78)
  at
com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:212)
  at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:68)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at com.intellij.rt.execution.application.AppMain.main(AppMain.java:140)
```

...then you failed to set the `sdk` property to a valid API level in your `@Config` annotation, as is described earlier in this chapter.

OK, So, Why Bother?

If input is limited and mocks are incomplete, why are we using Robolectric?

Robolectric can be useful in cases where you wish to run a quick subset of tests frequently to determine whether there are egregious problems. So-called “smoke tests” might be run on every commit to a version control system by a build server, for example. In this case, while Roboletric may not help you with test breadth or depth, it can help you with test frequency, since unit tests run so much faster. Running your full test suite on every commit might be too much; running the subset that you can implement as unit tests may be far more practical.

MonkeyRunner and the Test Monkey

Many GUI environments have some means or another of "fuzz" or "bash" testing, where some test driver executes a bunch of random input, in hopes of catching errors (e.g., missing validation logic). Android offers the Test Monkey for this.

Many GUI environments have some means or another of scripting GUI events from outside the application itself, to simulate button clicks or touch events. Android offers MonkeyRunner for this.

As the names suggest, there is a bit of commonality in their implementation. And, as you might expect, there is a bit of commonality in their coverage in this book — we will examine both MonkeyRunner and the Test Monkey in this chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

MonkeyRunner

MonkeyRunner is a means of creating test suites for Android applications based on scripted UI input. Rather than write a series of JUnit test cases or the like, you create Jython (JVM implementation of Python) scripts that run commands to install apps, execute GUI events, and take screenshots of results.

Writing a MonkeyRunner Script

The primary object you will work with in a MonkeyRunner script is a `MonkeyDevice`, which represents your connection to the device or emulator that you are testing. You obtain a `MonkeyDevice` by calling `waitForConnection()` on `MonkeyRunner`; this will return once it has established a connection.

From there, `MonkeyDevice` lets you send events to the device or emulator:

- `installPackage()` allows you to install an APK from your development machine, and `removePackage()` allows you to get rid of it
- `startActivity()` and `broadcastIntent()` allow you to start up components of your app
- `press()` to simulate key events, including QWERTY keys, standard device keys like BACK, D-pad/trackball events, and anything else represented by a standard Android `KeyEvent`
- `type()` to simulate entering a whole string, as a simplification over calling `press()` once per letter
- `touch()` and `drag()` let you simulate touch events
- and so on

The biggest limitation is in getting data out of the device, to determine if your test worked successfully. Your options are:

- `takeSnapshot()`, which will capture a screenshot that you can save to disk, compare with other screenshots, etc.
- `shell()` executes `adb shell` commands, returning any results
- ...and that's about it

Unlike JUnit-based testing, you have no visibility into the activity beyond what appears on the screen — you cannot inspect widgets, call methods, or the like.

For example, here is a script that installs an app, runs an activity from it, and presses the down button on the D-pad three times:

```
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

device = MonkeyRunner.waitForConnection()
device.installPackage('bin/JUnitDemo.apk')
device.startActivity(component='com.commonsware.android.abf/
com.commonsware.android.abf.ActionBarFragmentActivity')
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
```

```
# result = device.takeSnapshot()
# result.writeToFile('tests/monkey_sample_shots/test1.png', 'png')
```

(from Testing/monkey_sample.py)

Executing MonkeyRunner

To execute your MonkeyRunner script, have your device or emulator set up at a likely starting point (e.g., home screen), then execute the `monkeyrunner` command, passing it the path to your script (e.g., `monkeyrunner monkey_sample.py`). You will see the script executing on the screen of your device or emulator, and your console will contain whatever output you might emit from your test script itself. For example, you might take screenshots, compare them against a master copy (using methods on `MonkeyImage` to help with this), and emit warnings if they differ unexpectedly.

Monkeying Around

Independent from the JUnit system and MonkeyRunner is the Test Monkey (referred to here as "the Monkey" for short).

The Monkey is a test program that simulates random user input. It is designed for "bash testing", confirming that no matter what the user does, the application will not crash. The application may have odd results — random input entered into a Twitter client may, indeed, post that random input to Twitter. The Monkey does not test to make sure that results of random input make sense; it only tests to make sure random input does not blow up the program.

You can run the Monkey by setting up your initial starting point (e.g., the main activity in your application) on your device or emulator, then running a command like this:

```
adb shell monkey -p your.package.here -v --throttle 100 600
```

(substituting the application ID of a project on your device or emulator for `your.package.here`)

Working from right to left, we are asking for 600 simulated events, throttled to add 100 millisecond delays. We want to see a list of the invoked events (`-v`) and we want to throw out any event that might cause the Monkey to leave our application, as determined by the application ID (`-p your.package.here`).

Note that this truly is the application ID, not the package name. In simple apps — such as most of this book's samples — the package name *is* the application ID. But if you have modified the application ID in your `build.gradle` file (e.g., replaced the application ID for a product flavor), you will need to use the actual application ID here, not the package name. In general, despite any legacy documentation to the contrary, "package name" only affects code-generated classes like `R` and `BuildConfig`. Any other use of the term "package name" really is referring to the application ID.

The Monkey will simulate keypresses (both QWERTY and specialized hardware keys, like the volume controls), D-pad/trackball moves, and sliding the keyboard open or closed. Note that the latter may cause your emulator some confusion, as the emulator itself does not itself actually rotate, so you may end up with your screen appearing in landscape while the emulator is still, itself, portrait. Just rotate the emulator a couple of times (e.g., `<Ctrl>-<F12>`) to clear up the problem.

Also note that the throttle time is only used in between batches of related events. So, a batch of several touch events, or a pair of up/down events for a hardware key, will not be throttled. You can see this if you pass `-v -v -v` for "ultimate verbose mode" and look at the output, such as this snippet:

```
:Sending Key (ACTION_DOWN): 134    // KEYCODE_F4
:Sending Key (ACTION_UP): 134    // KEYCODE_F4
Sleeping for 100 milliseconds
:Sending Touch (ACTION_DOWN): 0:(1302.0,842.0)
:Sending Touch (ACTION_MOVE): 0:(1295.3395,838.0863)
:Sending Touch (ACTION_MOVE): 0:(1290.1539,827.0493)
:Sending Touch (ACTION_MOVE): 0:(1280.0454,826.9068)
:Sending Touch (ACTION_MOVE): 0:(1272.0161,816.8062)
:Sending Touch (ACTION_MOVE): 0:(1260.7244,810.8302)
:Sending Touch (ACTION_UP): 0:(1250.5455,801.67444)
Sleeping for 100 milliseconds
:Sending Key (ACTION_DOWN): 19    // KEYCODE_DPAD_UP
:Sending Key (ACTION_UP): 19    // KEYCODE_DPAD_UP
Sleeping for 100 milliseconds
:Sending Key (ACTION_DOWN): 68    // KEYCODE_GRAVE
:Sending Key (ACTION_UP): 68    // KEYCODE_GRAVE
Sleeping for 100 milliseconds
```

Here, we see actual messages where the throttling is applied ("Sleeping for 100 milliseconds"). Hence, the time it takes the Test Monkey to run a test will be at most the number of events times the throttle time, but due to event batching, it is usually substantially less than that. 25-30% of the maximum time seems typical.

For playing with a Monkey, the above command works fine. However, if you want to regularly test your application this way, you may need some measure of repeatability. After all, the particular set of input events that trigger your crash may not come up all that often, and without that repeatable scenario, it will be difficult to repair the bug, let alone test that the repair worked.

To deal with this, the Monkey offers the `-s` switch, where you provide a seed for the random number generator. By default, the Monkey creates its own seed, giving totally random results. If you supply the seed, while the sequence of events is random, it is random for that seed — repeatedly using the same seed will give you the same events. If you can arrange to detect a crash and know what seed was used to create that crash, you may well be able to reproduce the crash.

Trail: Rx

Java 8 Lambda Expressions

In 2016, Android app development gained some ability to use Java 8 programming constructs.

One of those changes was the release of Android 7.0, which introduced some Java 8-compatible classes, such as those in the `java.util.stream` package. These are new to API Level 24. Most likely, you will only start to use them once you raise your `minSdkVersion` to 24 or higher.

However, some features can be used on older devices. Notable among these are lambda expressions, the Java 8 equivalent of blocks or closures that you find in other programming languages. Lambda expressions can make your code a bit less verbose, particularly in places where you are making heavy use of listener interfaces or other forms of callback objects.

Getting all this to work requires some Gradle changes (to request Java 8 support in the build process), plus using the new lambda expression syntax. This chapter will show you all of this.

Prerequisites

Understanding this chapter requires that you have read the core chapters of the book. It does not require that you have prior experience with Java 8 lambda expressions.

However, having read the chapter on `RecyclerView` is a good idea, as the sample app in this chapter was originally shown there.

Also, you should now be using Android Studio 3.0, and its related build tools.

The Basic Idea

In Java programming prior to Java 8, we needed to create a lot of classes for minor roles, whether those classes were traditional Java classes, nested classes, or anonymous inner classes.

For example, suppose that you have an `ArrayList` of `Video` objects:

```
package com.commonsware.android.recyclerview.videolist;

import android.content.ContentUris;
import android.database.Cursor;
import android.net.Uri;
import android.provider.MediaStore;

class Video implements Comparable<Video> {
  final String title;
  final Uri videoUri;
  final String mimeType;

  Video(Cursor row) {
    this.title=
      row.getString(row.getColumnIndex(MediaStore.Video.Media.TITLE));
    this.videoUri=ContentUris.withAppendedId(
      MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
      row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID)));
    this.mimeType=
      row.getString(row.getColumnIndex(MediaStore.Video.Media.MIME_TYPE));
  }

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Video)) {
      return(false);
    }

    return(videoUri.equals(((Video)obj).videoUri));
  }

  @Override
  public int hashCode() {
    return(videoUri.hashCode());
  }

  @Override
  public int compareTo(Video video) {
```

```
    return(title.compareTo(video.title));
  }
}
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/Video.java)

You want to sort those videos by a custom algorithm: ascending by title, descending by title, etc.

That is easy enough with `Collections.sort()`:

```
Collections.sort(temp, new Comparator<Video>() {
  @Override
  public int compare(Video one, Video two) {
    return(one.compareTo(two));
  }
});
```

However, that is fairly verbose. Many languages offer a simpler syntax, where you can provide a "block" or "closure" that, in this case, would handle the comparison without having to create a subclass and implement a method.

Java 8 offers such a syntax, via lambda expressions:

```
        Collections.sort(newVideos,
          (one, two) -> one.compareTo(two));
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

Here, we have replaced the `Comparator` anonymous inner class implementation with a single line of Java code, but having the same result. The compiler, under the covers, wraps your lambda expression in a suitable `Comparator` for use by the `sort()` method.

Using Lambda Expressions

With all that as prologue, let's look at how we can enable and employ lambda expressions in an Android application.

The code listings for this section come mostly from the `Java8/VideoLambda` sample project. This is a version of a sample app from the chapter on `RecyclerView`, where we query the `MediaStore` for available videos and show them in a list. There are a

couple of editions of that sample in the `RecyclerView` chapter, one of which shows the use of `DiffUtil`, and that sample also demonstrates lambda expressions.

Enabling Lambda Expressions

Prior to Android Studio 3.0 (and its associated build tools), lambda expression support was offered by Google through a different Java compiler, called Jack. Jack had issues, possibly due to a broken crown. Android Studio 3.0 eliminated Jack, offering lambda expression support by means of the classic `javac` compiler combined with updated version of the `dx` tool for creating Dalvik bytecode from Java bytecode.

So, nowadays, to use lambda expressions, so long as you are using current build tools, the only project change that you need is to indicate that your source and target are both Java 1.8:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.android.support:recyclerview-v7:26.1.0'
    implementation 'com.squareup.picasso:picasso:2.5.2'
}

android {
    compileSdkVersion 26

    defaultConfig {
        minSdkVersion 16
        targetSdkVersion 25
        applicationId 'com.commonsware.android.lambda.video'
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

(from Java8/VideoLambda/app/build.gradle)

Replacing Listeners with Lambdas

Now, you can replace single-method anonymous inner class implementations with lambdas with relative ease.

The example lambda expression from earlier in this chapter comes from the `sortAndApply()` method of the `VideoAdapter` inside the `MainActivity` of the sample app. Given an `ArrayList` of `Video` objects, we need to sort them based on the user's requested sort order (ascending by default). We use lambda expressions, rather than custom `Comparator` anonymous inner classes, to do that sorting:

```
  private void sortAndApply(ArrayList<Video> newVideos) {
    if (sortAscending) {
      Collections.sort(newVideos,
        (one, two) -> one.compareTo(two));
    }
    else {
      Collections.sort(newVideos,
        (one, two) -> two.compareTo(one));
    }

    DiffUtil.Callback cb=new SimpleCallback<>(videos, newVideos);
    DiffUtil.DiffResult result=DiffUtil.calculateDiff(cb, true);

    videos=newVideos;
    result.dispatchUpdatesTo(this);
  }
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

Here, each of our two lambda expressions:

- Has two parameters (the two objects passed into `compare()` of a `Comparator`), so they get wrapped in parentheses
- Has a single Java statement to be executed, so it stands alone after the arrow (`->`)

Other editions of this sample app have `RowController` implement the `OnClickListener` interface and implement `onClick()` to be able to respond to click events on our rows. This sample app uses a lambda expression instead:

```
package com.commonsware.android.recyclerview.videolist;

import android.content.ContentUris;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.provider.MediaStore;
import android.support.v7.widget.RecyclerView;
import android.view.View;
```

```
import android.widget.ImageView;
import android.widget.TextView;
import com.squareup.picasso.Picasso;

class RowController extends RecyclerView.ViewHolder {
  private TextView title=null;
  private ImageView thumbnail=null;
  private Uri videoUri=null;
  private String videoMimeType=null;

  RowController(View row) {
    super(row);

    title=row.findViewById(android.R.id.text1);
    thumbnail=row.findViewById(R.id.thumbnail);

    row.setOnClickListener(view -> {
      Intent i=new Intent(Intent.ACTION_VIEW);

      i.setDataAndType(videoUri, videoMimeType);
      title.getContext().startActivity(i);
    });
  }

  void bindModel(Video video) {
    title.setText(video.title);

    videoUri=video.videoUri;
    videoMimeType=video.mimeType;

    Picasso.with(thumbnail.getContext())
      .load(videoUri.toString())
      .fit().centerCrop()
      .placeholder(R.drawable.ic_media_video_poster)
      .into(thumbnail);
  }
}
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/RowController.java)

Here, our lambda expression:

- Has one parameter, so we can skip the parentheses around it
- Has multiple Java statements to be executed, so they get wrapped in a standard Java block ({ and })

When Android Studio detects a place where you could use a lambda expression, it will mark the anonymous inner class as gray:

```
row.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View view) {
    Intent i=new Intent(Intent.ACTION_VIEW);

    i.setDataAndType(videoUri, videoMimeType);
    title.getContext().startActivity(i);
  }
});
```

Figure 375: Potential Lambda Expression, in Android Studio

A quick-fix (e.g., Alt-Enter on Windows and Linux) will offer to convert it into a lambda expression for you.

Alternative: Method References

Our `Video` model class has a `compareTo()` implementation that our sorting lambda expressions rely upon:

```
package com.commonsware.android.recyclerview.videolist;

import android.content.ContentUris;
import android.database.Cursor;
import android.net.Uri;
import android.provider.MediaStore;

class Video implements Comparable<Video> {
  final String title;
  final Uri videoUri;
  final String mimeType;

  Video(Cursor row) {
    this.title=
      row.getString(row.getColumnIndex(MediaStore.Video.Media.TITLE));
    this.videoUri=ContentUris.withAppendedId(
      MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
      row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID)));
    this.mimeType=
      row.getString(row.getColumnIndex(MediaStore.Video.Media.MIME_TYPE));
  }
```

```
  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Video)) {
      return(false);
    }

    return(videoUri.equals(((Video)obj).videoUri));
  }

  @Override
  public int hashCode() {
    return(videoUri.hashCode());
  }

  @Override
  public int compareTo(Video video) {
    return(title.compareTo(video.title));
  }
}
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/Video.java)

We can further simplify the sorting code by replacing the lambda expressions with a method reference... for the first such expression:

```
if (sortAscending) {
  Collections.sort(temp, Video::compareTo);
}
else {
  Collections.sort(temp,
    (one, two) -> two.compareTo(one));
}
```

Here, we are saying that we want to pass the parameters normally passed into `compare()` of a `Comparator` into the `compareTo()` method of the first parameter. You can similarly use method references to refer to static methods or a method on some separate object.

However, we cannot replace the second lambda expression with a method reference. The `Video::compareTo` method reference always calls the method on the first parameter. In our case, that is fine for the ascending sort, but for the descending sort, we would need to call `compareTo()` on the *second* parameter, as our lambda expression does. Alternatively, `Video` could implement another method (e.g., `compareDescendingTo()`) that we could use as a method reference.

Rx Basics

A programming model that has been gaining substantial ground in recent years is reactive programming. In Android, much of the attention has been on RxJava, the "reactive extensions for Java". Many libraries offer the ability to be consumed using RxJava, and many Android experts have latched onto RxJava as a way to reduce certain types of complexity in Android app development.

In this chapter, we will review what reactive programming is, what RxJava is, and how you can apply RxJava in your Android app.

However, please understand that reactive programming is a very large topic. Just a complete explanation of RxJava would entail its own book. This chapter should be seen as a launching pad for further explorations of your own, more so than a definitive reference on the subject.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Life is But a Stream

In order to understand reactive programming, we first need to think about streams.

When a Java programmer hears the term "stream", what often pops into mind is `InputStream` and `OutputStream`. Those offer access to a stream of bytes, for input and output, respectively. Here, "stream" means that the bytes are available one at a time (though are often retrieved in a clump, such as an 8192-byte buffer), and that

once removed from the stream the bytes are considered to be "consumed" and are no longer available from the stream itself.

When Java programmers think of `InputStream` and `OutputStream`, what often pops into mind is `FileInputStream` and `FileOutputStream`. With `FileInputStream`, the source of the bytes is fixed: the contents of a designated file. With `FileOutputStream`, the destination of the bytes is fixed: once again, a designated file.

However, there are many other sources of `InputStream` and `OutputStream`. Some that you encounter in the book are:

- Streams on sockets, such as the `InputStream` that you get from `HttpUrlConnection`
- Streams on content from a `ContentProvider`, such as the `InputStream` that you get from calling `openInputStream()` on a `ContentProvider`

Particularly in the HTTP case, the source of the bytes is "live", insofar as there does not have to be some specific file that is the source of those bytes. Those bytes could represent a generated Web page, or a live audio stream, or anything else.

Hence, more generally, a stream represents a flow of data, where we can pull data off of the stream and do something with it. That "flow of data" could be bytes from a file, as we see with `InputStream`. But lots of other things could be modeled as flows of data. Pretty much anything where the data would come to us asynchronously could be modeled this way, such as:

- Sensor readings off of an accelerometer
- GPS fixes
- Touch events
- Audio signals from a microphone
- Preview or live video frames from a camera
- And so on

You could even model some things that might not feel like a "stream" as a stream if you wanted to. For example, querying a database or `ContentProvider` gives you a `Cursor` back, and you could model that as being a stream of rows.

Action and Reaction

Reactive programming works with streams. Rather than "iterate over the rows in the `Cursor` and do X to each", you say "as a row comes in from the stream of rows, do X to each". Here, you are "reacting" to the availability of a row.

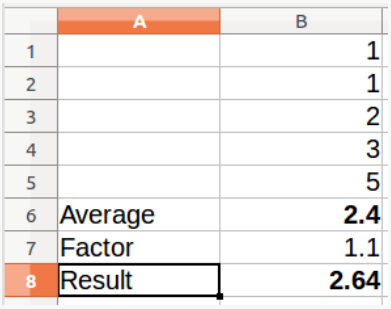
But part of reactive programming is that what *processes* a stream might itself turn around and *be a stream* of events to something else. For example, given a stream of audio signals, you might implement a filter that clamps high-volume signals to a lower volume. That filter takes in a stream (raw signals) and emits another stream (clamped signals). Other code could react to the filtered stream (e.g., to record bytes to disk).

You may have already done reactive programming, without even realizing it. The world's most popular reactive programming environment is not some functional programming language or UI framework.

It's a spreadsheet.

Spreadsheet cells — particularly for simpler sheets — can be thought of as either having data (particularly numbers) or formulas. A formula references cells and performs a calculation upon them. This includes performing calculations upon other cells that contain formulas.

For example, you might have a spreadsheet like this:



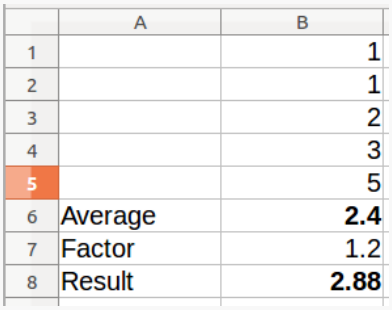
	A	B
1		1
2		1
3		2
4		3
5		5
6	Average	**2.4**
7	Factor	1.1
8	Result	**2.64**

Figure 376: Simple Spreadsheet

Here, the two formulas are shown in bold, while the data cells are in the default font.

The average formula cell has `=AVERAGE(B1:B5)`, to compute the average of the five number cells above it. The result formula cell has `=B6*B7`, to multiply the average by the "factor" number.

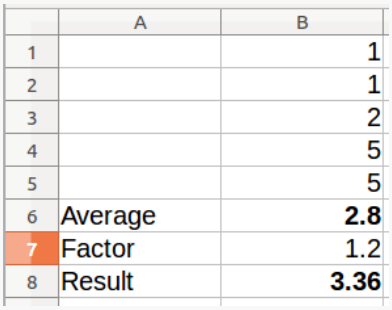
If you change that factor, not only does that cell change, but so does the result formula cell. The B6 and B7 values referenced in the result formula do not just identify cells, but represent *streams of changes to those cells' values*. When B7 (the factor value) changes, the result formula reacts to the changed-value event and recalculates its formula, showing the result:



	A	B
1		1
2		1
3		2
4		3
5		5
6	Average	**2.4**
7	Factor	1.2
8	Result	**2.88**

Figure 377: Simple Spreadsheet with New Factor Value

Similarly, if you change one of the five initial values (1, 1, 2, 3, 5), the average formula cell responds to the changed-value event and recalculates its value. That, in turn, triggers a changed-value event that causes the result formula cell to react and recalculate *its* value:



	A	B
1		1
2		1
3		2
4		5
5		5
6	Average	**2.8**
7	Factor	1.2
8	Result	**3.36**

Figure 378: Simple Spreadsheet with New Initial and New Factor Value

A Rx For What Ails You

At this point, you may be wondering what this has to do with Android, or even Java.

The pre-eminent framework for reactive programming in Java — while sticking to Java syntax itself, versus using a functional JVM language like Scala or Clojure — is RxJava. RxJava is a library that helps you model data as streams and apply a chain of operations to those streams.

RxJava is part of the ReactiveX series of libraries, offering reactive programming for a wide range of programming languages and platforms.

In late 2016, RxJava 2 was released. This includes some refactoring of the original RxJava classes to comply with the Reactive Streams specification. This chapter focuses on RxJava 2. Some material that you read on RxJava might be using the original RxJava API. While pretty much everything done with classic RxJava can be done with RxJava2, some conversion may be required.

(BTW, for those of you wondering about the title of this section, Rx is an abbreviation for the word "prescription")

Rx and Lambdas

RxJava relies heavily on functions. Java is not a functional programming language, and so it does not have "functions" as standalone first-class units of programming.

Java 8 offers lambda expressions, which are akin to anonymous functions. RxJava supports Java 8 lambda expressions, and by using Android Studio 3.0 (and its related build tools), Android developers can use lambda expressions in projects with a `minSdkVersion` of 9 or higher.

The examples in this chapter use lambda expressions, as that will be the typical way that you see RxJava be used.

A Simple Stream

Back in the chapter on threads, we had the `Threads/AsyncTask` sample app, which demonstrated using an `AsyncTask`. There, our data source was a `static` Java array of strings, and we would not need an `AsyncTask` to "load" those into a list. However, to keep things simple, we pretended that it took real work to obtain those strings, and we used an `AsyncTask` to do that work in the background.

Let's explore reactive programming and RxJava by revamping that example, starting with the `Rx/Simple` sample project.

Adding the Dependency

RxJava is a library, available from Maven Central and JCenter. So, in a typical Android Studio project could add RxJava as a dependency in a module's `build.gradle` file:

```
apply plugin: 'com.android.application'

dependencies {
  implementation "com.android.support:support-fragment:27.0.2"
  implementation 'io.reactivex.rxjava2:rxjava:2.1.7'
}

android {
  compileSdkVersion 27
  buildToolsVersion '27.0.3'

  defaultConfig {
    minSdkVersion 15
    targetSdkVersion 27
  }

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}
```

(from Rx/Simple/app/build.gradle)

Replacing the Loop

The original `AsyncTask` sample app has a `doInBackground()` method that looks a bit like this:

```
@Override
protected Void doInBackground(Void... unused) {
  for (String item : ITEMS) {
    if (isCancelled())
      break;

    publishProgress(item);
    SystemClock.sleep(400);
  }

  return(null);
}
```

Here, `ITEMS` is our `static` String array. We iterate over that array, and if our task has not been canceled, we call `publishProgress()` to add a string to our `ListView`, and sleep for 400 milliseconds as our way of pretending to do real work.

The first step towards switching to a reactive model is to replace that loop with a stream:

```
  @Override
  protected Void doInBackground(Void... unused) {
    Observable.fromArray(ITEMS).subscribe(s -> {
      if (!isCancelled()) {
        publishProgress(s);
        SystemClock.sleep(400);
      }
    });

    return(null);
  }
```

(from Rx/Simple/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

`io.reactivex.Observable` is the root class for most of what you will do with RxJava. It helps you set up a stream and then react to that stream.

There are several ways to set up a stream — we will see a few of them in this chapter. In this case, we are using `fromArray()`, which sets up a stream based on the items in that array. In reality, this is a factory method; the returned object is an `Observable`.

We then call `subscribe()` on the `Observable`. This basically represents the sink for the stream of data coming from the `Observable`. In this case, we use lambda expression implementation of the `Consumer` interface. Our lambda expression body implements the `accept()` method from that `Consumer` interface, and it will be invoked for each piece of data from the stream. In this case, `ITEMS` is an array of `String`, and so we are setting up a `Consumer` of `String` and therefore we `accept()` a `String`. Our lambda expression has our `isCancelled()` check, our `publishProgress()` call, and our `sleep()` call, as with the iterative loop from the original sample.

Be Your Own Stream

Our 400 milliseconds of `sleep()` is a way of simulating doing work. However, as RxJava is showing us, we are not actually simulating doing work to get the data.

Instead, we are simulating doing work to *consume* the data. Our `sleep()` call is in our `Consumer` (or the equivalent lambda expression), not in our data source.

However, we can model this better, by creating our own data source that is a bit more sophisticated than a simple `static` Java array.

The `Rx/Observable` sample project replaces `fromArray()` with a call to `create()`, which creates an `Observable` upon a supplied source of data:

```
@Override
protected Void doInBackground(Void... unused) {
  Observable.create(source())
    .subscribe(s -> {
      if (!isCancelled()) {
        publishProgress(s);
      }
    });

  return(null);
}

private ObservableOnSubscribe<String> source() {
  return(new ObservableOnSubscribe<String>() {
    @Override
    public void subscribe(ObservableEmitter<String> emitter) throws Exception {
      for (String item : ITEMS) {
        if (!isCancelled()) {
          emitter.onNext(item);
          SystemClock.sleep(400);
        }
      }

      emitter.onComplete();
    }
  });
}
```

Here, `source()` is returning an instance of an oddly-named `ObservableOnSubscribe` interface. An `ObservableOnSubscribe` is for cases where we are only actually starting to obtain data for our stream when something subscribes to the `Observable`, not before. In our case, that is fine, as we are still just using our `String` array.

An `ObservableOnSubscribe` implements a `subscribe()` method. This is called when a subscriber subscribes to the `Observable`. The job of `subscribe()` is to work with the underlying data and call three methods on the supplied `ObservableEmitter`:

- Call `onNext()`for each piece of data in the stream
- Call `onComplete()` when we are done with all possible data
- Call `onError()` if something fails (e.g., an `IOException` when reading from a socket)

In our case, we are iterating over our Java array, calling `onNext()` for each `String`, and sleeping the 400 milliseconds to simulate doing actual work.

Note that, if you want, you can replace the `ObservableOnSubscribe` with a lambda expression:

```
  private ObservableOnSubscribe<String> source() {
    return(emitter -> {
      for (String item : ITEMS) {
        if (!isCancelled()) {
          emitter.onNext(item);
          SystemClock.sleep(400);
        }
      }

      emitter.onComplete();
    });
  }
```

(from Rx/Observable/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

Removing the AsyncTask

At this point, you may be wondering why we are bothering with any of this, as our code gets more and more complex without any particular added value. This reaction is understandable. All we have done is make an `AsyncTask` more complicated.

So, let's get rid of the `AsyncTask`.

Part of the power of RxJava is thread management. You can tell it the threads to use for working with streams, such as doing I/O on a background thread. However, RxJava is a pure Java library. It knows nothing about Android and Android-specific constructs like the main application thread.

Fortunately, there is RxAndroid. This a small library adding some Android-specific smarts to RxJava, such as the notion of the main application thread. This, like RxJava, is available as an artifact:

```
apply plugin: 'com.android.application'

dependencies {
  implementation "com.android.support:support-fragment:27.0.2"
  implementation 'io.reactivex.rxjava2:rxjava:2.1.7'
  implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'
}

android {
  compileSdkVersion 27
  buildToolsVersion '27.0.3'

  defaultConfig {
    minSdkVersion 15
    targetSdkVersion 27
    applicationId "com.commonsware.android.rx.rxandroid"
  }

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}
```

(from Rx/RxAndroid/app/build.gradle)

With that in mind, let's look at the Rx/RxAndroid sample project and our substantially-revised `ListFragment`, here named `RxDemoFragment`:

```
package com.commonsware.android.rx;

import android.support.v4.app.ListFragment;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Toast;
import java.util.ArrayList;
import io.reactivex.Observable;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.android.schedulers.AndroidSchedulers;
import io.reactivex.disposables.Disposable;
import io.reactivex.schedulers.Schedulers;

public class RxDemoFragment extends ListFragment {
  private static final String[] ITEMS= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
```

```
    "vel", "erat", "placerat", "ante", "porttitor", "sodales",
    "pellentesque", "augue", "purus" };
private ArrayList<String> model=new ArrayList<>();
private ArrayAdapter<String> adapter;
private Disposable sub=null;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  adapter=new ArrayAdapter<>(getActivity(),
    android.R.layout.simple_list_item_1, model);

  Observable<String> observable=Observable
    .create(source())
    .subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())
    .doOnComplete(() -> {
      Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
        .show();
    });

  sub=observable.subscribe(s -> {
    adapter.add(s);
  });
}

private ObservableOnSubscribe<String> source() {
  return(emitter -> {
    for (String item : ITEMS) {
      emitter.onNext(item);
      SystemClock.sleep(400);
    }

    emitter.onComplete();
  });
}

@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  getListView().setScrollbarFadingEnabled(false);
  setListAdapter(adapter);
}
```

```
  @Override
  public void onDestroy() {
    if (sub!=null && !sub.isDisposed()) {
      sub.dispose();
    }

    super.onDestroy();
  }
}
```

(from Rx/RxAndroid/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

Some things are the same as in the original sample app: the `static` array, the `ArrayList` model, the `ArrayAdapter`, the retained fragment, etc.

However, our `AsyncTask` is gone, replaced by a chunk of RxJava code:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);

    adapter=new ArrayAdapter<>(getActivity(),
      android.R.layout.simple_list_item_1, model);

    Observable<String> observable=Observable
      .create(source())
      .subscribeOn(Schedulers.newThread())
      .observeOn(AndroidSchedulers.mainThread())
      .doOnComplete(() -> {
        Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
          .show();
      });

    sub=observable.subscribe(s -> {
      adapter.add(s);
    });
  }
```

(from Rx/RxAndroid/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

This code adds a number of items to our `Observerable` call chain, so let's take them one at a time.

subscribeOn()

This teaches RxJava the code to use for invoking our `ObservableOnSubscribe` code, specifically its `subscribe()` method (or the equivalent with a lambda expression). Since `subscribe()` could take some time — in our case, 400 milliseconds per string — we really want to run that on a background thread.

The `Schedulers` utility class in RxJava provides a handful of options for indicating the thread to use. Here, we are using `newThread()`, which does pretty much what the name suggests: create a new thread and use that for our `subscribe()` work.

Typically, if we are using RxJava and RxAndroid, we want to subscribe to events on a background thread. That is not always the case, though. There are techniques for using RxJava/RxAndroid to manage things like user input (e.g., text being entered live in an `EditText`). In those cases, we may have specific threading requirements, such as subscribing on Android's main application thread.

observeOn()

By default, if you use `subscribeOn()`, *everything* involved with this stream will be done on that thread. This would include the work we want to do to update our UI, via our `subscribe()` call. And, on Android, we cannot update the UI from a background thread.

`observeOn()` basically says "for everything in this `Observable` configuration from this point forward should switch to using this other thread". With our `subscribe()` call coming after `observeOn()`, we wind up processing our `subscribe()` work in the thread indicated by `observeOn()`.

Here, we use RxAndroid and its `AndroidSchedulers` class, which has a `mainThread()` method to indicate that we want the work to be done on the main application thread.

Note that you can call `observeOn()` several times. Typically, this is not necessary. But, for example, you might use `subscribeOn()` to receive some UI event on the main application thread, then call `observeOn()` to direct initial processing of that event onto a background thread (e.g., database I/O for updating a live filter based on what has been typed in so far), then call `observeOn()` *again* to direct further processing to the main application thread (e.g., update a `RecyclerView` with the now-filtered results).

doOnComplete()

In our samples so far, in the `onPostExecute()` method of our `AsyncTask`, we showed a `Toast` to signal that the work was completed.

The equivalent step in RxJava is `doOnComplete()`. This takes an `Action` object or, in this case, a lambda expression. This will be called when our `source()` triggers `onCompleted()` on its supplied `emitter`. Here, we show the `Toast`, after confirming that we have not been canceled.

subscribe()

We then `subscribe()` to the `Observable`, feeding the strings into our `ArrayAdapter`. `subscribe()` returns a `Disposable` instance, which we hold onto in a field.

On a configuration change, we retain the `Disposable`, which in turn holds onto our `Observable`. We also retain the `ArrayAdapter`. So, the new activity and new fragment connect back up to the `ArrayAdapter`, and they get all existing words plus any new ones fed in by our ongoing Rx work, if that has not completed already.

If the user presses BACK to exit the activity, or the activity otherwise is being finally destroyed, `onDestroy()` is called on our retained fragment, and there we `dispose()` of that `Disposable`:

```
@Override
public void onDestroy() {
  if (sub!=null && !sub.isDisposed()) {
    sub.dispose();
  }

  super.onDestroy();
}
```

(from Rx/RxAndroid/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

This way, if the Rx background work is still going on, and the fragment is being destroyed, we disconnect and do not try to continue updating the `ArrayAdapter` or display the `Toast`, as neither of those things are safe with a destroyed activity.

Lambdas and Lifetimes

Be careful about what you reference from lambda expressions tied into RxJava, or their anonymous inner class counterparts. You need to ensure that the lambdas only reference objects with the same intended lifespan as the lambda itself.

In particular, be careful about referencing widgets directly. Suppose that we had some RxJava work that was delivering a `String` to be filled into an `EditText`. From a retained fragment, you might try doing something like this:

```
Observable<String> observable=Observable
  .create(source())
  .subscribeOn(Schedulers.newThread())
  .observeOn(AndroidSchedulers.mainThread());

sub=observable.subscribe(s -> {
  myAwesomeEditText.setText(s);
});
```

The problem here is that `myAwesomeEditText` is a specific `EditText` instance, the one known about at the time when the lambda expression is created. If we undergo a configuration change, our fragment winds up retaining the lambda expression *but creating a new `EditText`*. However, our lambda expression does not know about that new `EditText`, so it happily sets the text of the old `EditText`, with unfortunate results.

A safer approach is to call a method:

```
Observable<String> observable=Observable
  .create(source())
  .subscribeOn(Schedulers.newThread())
  .observeOn(AndroidSchedulers.mainThread());

sub=observable.subscribe(s -> {
  updateMyAwesomeEditText(s);
});
```

where the method on the fragment can refer to the proper `EditText` instance, whichever one is current at the time that the lambda expression is evaluated.

Streaming from a Resource

Of course, this still uses a 400-millisecond `sleep()` call to simulate real work. The next step would be to do some actual I/O here, taking advantage of RxJava's ability to pull data from the stream on a background thread.

The Rx/XML sample project places our 25 Latin words in an XML resource:

```
<words>
  <word value="lorem" />
  <word value="ipsum" />
  <word value="dolor" />
  <word value="sit" />
  <word value="amet" />
  <word value="consectetuer" />
  <word value="adipiscing" />
  <word value="elit" />
  <word value="morbi" />
  <word value="vel" />
  <word value="ligula" />
  <word value="vitae" />
  <word value="arcu" />
  <word value="aliquet" />
  <word value="mollis" />
  <word value="etiam" />
  <word value="vel" />
  <word value="erat" />
  <word value="placerat" />
  <word value="ante" />
  <word value="porttitor" />
  <word value="sodales" />
  <word value="pellentesque" />
  <word value="augue" />
  <word value="purus" />
</words>
```

(from Rx/XML/app/src/main/res/xml/words.xml)

We can then create an `ObservableOnSubscribe` implementation that reads in those words, using an `XmlPullParser` obtained from a `Resources` object, and use those words for our `onNext()` calls:

```
  private static class WordSource implements ObservableOnSubscribe<String> {
    private final Resources resources;
```

```
    WordSource(Context ctxt) {
      resources=ctxt.getResources();
    }

    @Override
    public void subscribe(ObservableEmitter<String> emitter) {
      try {
        XmlPullParser xpp=resources.getXml(R.xml.words);

        while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
          if (xpp.getEventType()==XmlPullParser.START_TAG) {
            if (xpp.getName().equals("word")) {
              emitter.onNext(xpp.getAttributeValue(0));
            }
          }

          xpp.next();
        }

        emitter.onComplete();
      }
      catch (Exception e) {
        emitter.onError(e);
      }
    }
  }
}
```

(from Rx/XML/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

Error Handling

If you look closely at that code snippet, you will see that we are using a third method on the `ObservableEmitter`: `onError()`. We call this when something goes wrong in reading in the XML, passing the exception along to the emitter.

That, in turn, can make it to the code in our `subscribe()` call:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  adapter=new ArrayAdapter<>(getActivity(),
    android.R.layout.simple_list_item_1, model);

  Observable<String> observable=Observable
    .create(new WordSource(getActivity()))
    .subscribeOn(Schedulers.io())
```

```
    .map(s -> (s.toUpperCase()))
    .observeOn(AndroidSchedulers.mainThread())
    .doOnComplete(() ->
      Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT).show());

  sub=observable.subscribe(s -> adapter.add(s),
      error ->
        Toast
          .makeText(getActivity(), error.getMessage(), Toast.LENGTH_LONG)
          .show());
}
```

(from Rx/XML/app/src/main/java/com/commonsware/android/rx/RxDemoFragment.java)

Now, we are using a two-parameter implementation of `subscribe()` and are passing in two lambda expressions. The first provides us our data, which we pass to the `ArrayAdapter` as before. The second lambda will be called if an exception is encountered and supplied to our `emitter` and its `onError()` method. Here, we get the error, and use its message to show a `Toast`.

Transmogrification

If you look closely at *that* code snippet, we have two other changes, compared to the earlier RxAndroid sample.

One is that we are using `Schedulers.io()`, instead of `Schedulers.newThread()`. RxJava has a thread dedicated for I/O concerns, which we are using here. That is not a requirement, and for more sophisticated scenarios you might need something else (e.g., a thread from a thread pool that you configure and manage). Here, we are just illustrating more than one way to move our I/O code off the main application thread.

The second change, while small here, is more profound in general. We are calling `map()`, passing in a lambda expression that takes a `String` and converts it to uppercase. If you run this sample app, you will see that all of the words show up in uppercase (and quickly, since reading words out of an XML resource happens much faster than one word every 400 milliseconds).

`map()` is known as an operator. Its job is to take a stream as input and emit another stream as output, executing some code on each item in the stream to change it, somehow.

There are *lots* of operators built into RxJava, handling all sorts of scenarios, such as filtering:

- `skip()` ignores a specified number of items, not passing them downstream
- `take()` accepts a specified number of items, ignoring the rest
- `distinct()` skips any items that appeared before, using `equals()` by default

The `takeUntil()` method that we have been calling is another operator, saying "take all items from the main `Observable` until this other `Observable` says otherwise".

Other operators aggregate or convert the stream, such as:

- `concat()` takes two `Observables` and emits the items from the first, followed by the items from the second
- `count()` counts the items and emits a single-item stream containing the count of the original stream's items
- `reduce()` applies some supplied lambda expression (or the equivalent) to calculate some result (e.g., an average), emitting a single-item stream containing that result

RxJava alone has perhaps a hundred operators, and you can create others if none of the built-in ones meet your needs.

Rx-Enabled Libraries

Many libraries either offer an Rx-compatible API as part of their base functionality or offer an Rx-compatible bridge between your code and some existing, non-Rx-compatible API.

For example, Retrofit offers an RxJava 2-compatible adapter for Retrofit 2.x.

The `Rx/Retrofit` sample project is a clone of the original Retrofit 2.x sample from the chapter on Internet access, revised to use RxJava bindings instead of Retrofit's built-in asynchronous options.

The dependencies now include the RxJava 2 adapter for Retrofit, plus RxAndroid — these, in turn, will pull in RxJava:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
    implementation 'com.squareup.retrofit2:adapter-rxjava2:2.3.0'
    implementation ("com.android.support:support-v4:27.0.2") {
```

```
        exclude group: 'com.android.support', module: 'support-media-compat'
    } // for https://issuetracker.google.com/issues/64909326
    implementation 'io.reactivex.rxjava2:rxjava:2.1.7'
    implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'
}

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
        versionCode 1
        versionName "1.0"
        applicationId "com.commonsware.android.rx.retrofit"
    }

    dataBinding {
        enabled = true
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

(from Rx/Retrofit/app/build.gradle)

Now, though, `StackOverflowInterface` can return an `Observable`, rather than a `Call`:

```
package com.commonsware.android.databind.basic;

import io.reactivex.Observable;
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.http.GET;
import retrofit2.http.Path;
import retrofit2.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  Observable<SOQuestions> questions(@Query("tagged") String tags);

  @GET("/2.1/questions/{ids}?site=stackoverflow")
```

```
  Observable<SOQuestions> update(@Path("ids") String questionIds);
}
```

(from Rx/Retrofit/app/src/main/java/com/commonsware/android/databind/basic/StackOverflowInterface.java)

This gives us an `Observable` on a stream, just as if we had called `Observable.fromArray()` or `Observable.create()`. In this case, the "stream" is a single-item stream, containing an `SOQuestions` payload, which is our parsed response from the Stack Exchange API.

That, in turn, allows our `QuestionsFragment` to use RxJava to arrange to perform the I/O on the `io()` thread, process the response on the Android main application thread, plus process our questions or any errors that may crop up:

```
    so.questions("android")
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(result1 -> {
          for (Item item : result1.items) {
            Question question=new Question(item);

            questions.add(question);
            questionMap.put(question.id, question);
          }

          setListAdapter(new QuestionsAdapter(questions));
        }, t -> {
          Toast.makeText(getActivity(), t.getMessage(),
            Toast.LENGTH_LONG).show();
          Log.e(getClass().getSimpleName(),
            "Exception from Retrofit request to StackOverflow", t);
        }
      );
```

(from Rx/Retrofit/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

Further Reading

The primary source of documentation on RxJava comes in the form of the GitHub repo's wiki.

JavaDocs of the RxJava 2 API are also available.

What About LiveData?

In 2017, Google introduced the Architecture Components: a set of libraries designed to offer higher-level abstractions around core architecture concerns. One piece of the Architecture Components is `LiveData`, which aims to offer a simpler approach to the same reactive pattern that is espoused in RxJava/RxAndroid.

`LiveData` is covered in the companion volume, "Android's Architecture Components".

Trail: Advanced UI

Notifications

Pop-up messages. Tray icons and their associated "bubble" messages. Bouncing dock icons. You are no doubt used to programs trying to get your attention, sometimes for good reason.

Your phone also probably chirps at you for more than just incoming calls: low battery, alarm clocks, appointment notifications, incoming text message or email, etc.

Not surprisingly, Android has a whole framework for dealing with these sorts of things, collectively called "notifications".

Prerequisites

Understanding this chapter requires you to have read the core chapters of the book.

What's a Notification?

A service, running in the background, needs a way to let users know something of interest has occurred, such as when email has been received. Moreover, the service may need some way to steer the user to an activity where they can act upon the event — reading a received message, for example. For this, Android supplies status bar icons, flashing lights, and other indicators collectively known as "notifications".

Your current phone may well have such icons, to indicate battery life, signal strength, whether Bluetooth is enabled, and the like. With Android, applications can add their own status bar icons, with an eye towards having them appear only when needed (e.g., a message has arrived).

Notifications will appear in one of two places. On most devices, they will appear in the status bar, on the top of the screen, left-aligned:

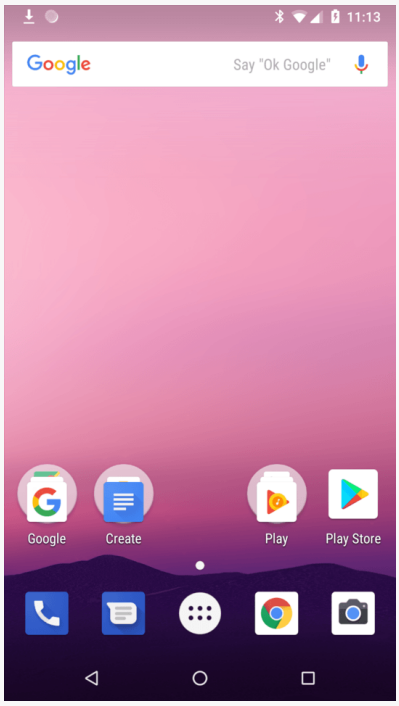


Figure 379: Notifications, on a Nexus 5X

On a pre-Android 4.2 tablet (and occasionally on other tablets newer than that), they will appear in the system bar, on the bottom of the screen, towards the lower-right corner:

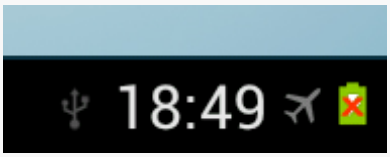


Figure 380: Notifications, on a Galaxy Tab 2

In either case, you can expand the “notification shade” to get more details about the active notifications, either by sliding down the status bar:

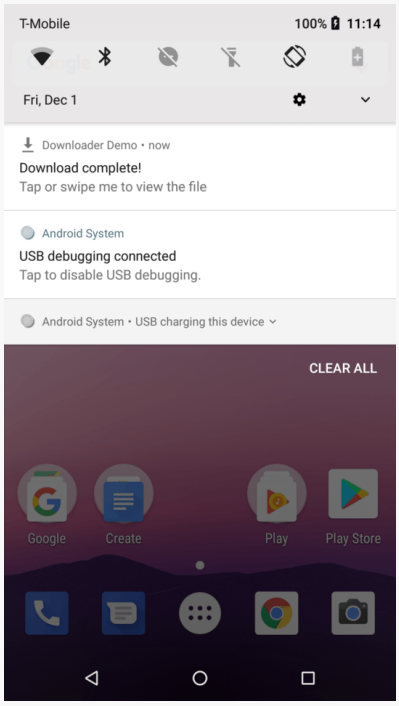


Figure 381: Notification Shade, on a Galaxy Nexus

or by tapping on the clock on the system bar on some tablets:

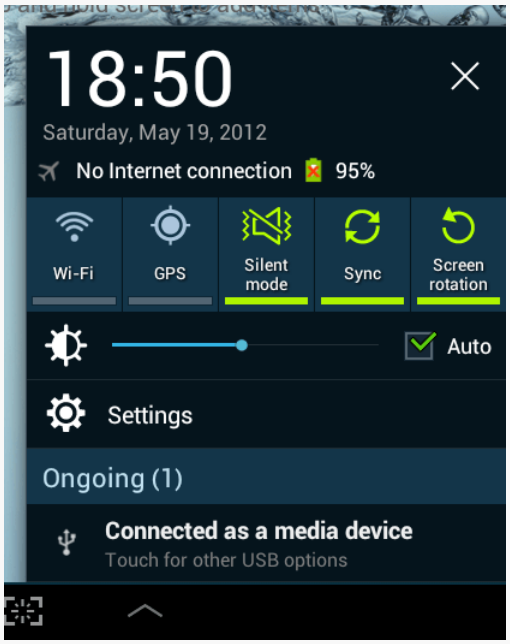


Figure 382: Notification Shade, on a Galaxy Tab 2

Some notifications will be complex, showing real-time information, such as the progress of a long download. More often, notifications are fairly simple, providing just a couple of lines of information, plus an identifying icon. Tapping on the notification shade entry will typically trigger some action, such as starting an activity — an email app letting the user know that "you've got mail" can have its notification bring up the inbox activity when tapped.

Notifications and Channels

Through Android 7.1, all notifications were created equal. In other words, the user could apply a single set of rules for configuring how those notifications were delivered from your app. How *much* configuration was even available varied by Android OS version, but on newer devices users could control whether to block those notifications, show them without any sound or vibration, what amount of information should be shown on the lockscreen, and so on.

In Android 8.0, all notifications go into channels that you define. The user then has the ability to control the behavior of notifications on a per-channel basis. This offers finer granularity of control over the breadth of notifications that your app might deliver. Android 8.0 also extends the depth of control, with more options for the user to decide how a channel's notifications should work.

Channels are not an optional feature, once your `targetSdkVersion` rises to `26` or higher. As a result, you will wind up with code that configures channels on API Level 26+ devices and code that configures notifications directly on older devices. This is aggravating, but is simply part of what is required.

In this chapter, we will show basic notification channel setup. In a later chapter, we will explore channels in greater detail.

Showing a Simple Notification

Suppose we want to download a file. That may take some time, depending on the size of the file. It would be nice to let the user know when the download has been completed. Ideally, we would let the user know by some means other than popping up a `Toast`. If we are having a service download the file — which is a good idea for longer downloads — there is the possibility that our UI is no longer in the foreground at the time the download is done, so we cannot necessarily update the UI to let the user know the file is ready for use.

An alternative would be for the background service doing the download to raise a `Notification` when the download is complete. That would work even if the activity was no longer around (e.g., user pressed BACK to exit it). This can be seen in the `Notifications/DownloadNotify` sample project. This is a slightly modified clone of the download-a-PDF-file sample from the chapter on services.

Our `DownloadFragment` for triggering the download dispenses with the `BroadcastReceiver` and logic related to it, including disabling and enabling the `Button`. Otherwise, it is the same as before.

The download logic in the `onHandleIntent()` method of `Downloader` is nearly identical as well, with two changes.

One change is that we pull out the MIME type of the response from its response header:

```
    URL url=new URL(i.getData().toString());
    HttpURLConnection c=(HttpURLConnection)url.openConnection();
    FileOutputStream fos=new FileOutputStream(output.getPath());
    BufferedOutputStream out=new BufferedOutputStream(fos);
    String mimeType=c.getHeaderField("Content-type");
```

(from Notifications/DownloadNotify/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

The other difference is that at the end, rather than sending a broadcast `Intent`, we call a private `raiseNotification()` method. We also call this method if there is an exception during the download. The `raiseNotification()` method takes the MIME type that we collected earlier, the `File` object representing the downloaded results (if we succeeded), and the `Exception` that was raised (if we crashed). As one might guess given the method's name, `raiseNotification()` will raise a `Notification`:

```
private void raiseNotification(String mimeType, File output,
                               Exception e) {
  NotificationManager mgr=
    (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
  }

  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);
```

```
    b.setAutoCancel(true);

    if (e == null) {
      b.setContentTitle(getString(R.string.download_complete))
       .setContentText(getString(R.string.fun))
       .setSmallIcon(android.R.drawable.stat_sys_download_done);

      Intent outbound=new Intent(Intent.ACTION_VIEW);
      Uri outputUri=
        FileProvider.getUriForFile(this, AUTHORITY, output);

      outbound.setDataAndType(outputUri, mimeType);
      outbound.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

      PendingIntent pi=PendingIntent.getActivity(this, 0,
        outbound, PendingIntent.FLAG_UPDATE_CURRENT);

      b.setContentIntent(pi);
    }
    else {
      b.setContentTitle(getString(R.string.exception))
       .setContentText(e.getMessage())
       .setSmallIcon(android.R.drawable.stat_notify_error);
    }

    mgr.notify(NOTIFY_ID, b.build());
  }
```

(from Notifications/DownloadNotify/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

The first thing we do in `raiseNotification()` is get a `NotificationManager`, which is a system service. Calling `getSystemService()` and asking for the `NOTIFICATION_SERVICE` will give us our `NotificationManager`, albeit after a cast.

Next, we need to consider setting up a notification channel. This is required on API Level 26+ devices, because our project's `targetSdkVersion` is 26. We need to give the channel some unique identifier, here labeled `CHANNEL_WHATEVER`:

```
  private static final String CHANNEL_WHATEVER="channel_whatever";
```

(from Notifications/DownloadNotify/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

`getNotificationChannel()`, on API Level 26+ devices, will return `null` if the channel is not defined. So, if we are on an API Level 26+ device and the channel is

not defined, we call `createNotificationChannel()` to define our channel. `createNotificationChannel()` takes three parameters:

- the channel identifier (`CHANNEL_WHATEVER`)
- a label to show the user that describes this channel (`"Whatever"`)
- the "importance" of this channel (`NotificationManager.IMPORTANCE_DEFAULT`)

We will cover the concept of "importance" in a later chapter. For now, take it on faith that `IMPORTANCE_DEFAULT` is a reasonable choice for most channels.

Then, we create a `NotificationCompat.Builder` object to help construct the `Notification`. On API Level 11 and higher, there is a `Notification.Builder` class that you can use. However, the notification system in Android has been changing frequently over the past few OS updates, and there are signs that this will continue. Hence, you may prefer to use `NotificationCompat.Builder`. First, this will work back to API Level 4, in case you are supporting notifications on older devices. More importantly, `NotificationCompat.Builder` is updated to reflect the latest `Notification.Builder` API, offering a backwards-compatible implementation of that API. Some newer features are not supported on older devices, but the `NotificationCompat.Builder` API lets you code to the new API, and it quietly ignores things that cannot be done on older devices.

The `NotificationCompat.Builder` constructor takes a `Context` plus the identifier of the channel that will be used. That channel identifier will be used on API Level 26+ devices and ignored otherwise. Here, we provide the `CHANNEL_WHATEVER` identifier that we just set up.

We can call methods on the `Builder` to configure the `Notification` that we want to display. Whether our download succeeded or failed, we call `setAutoCancel(true)` on the `Builder`, which means that when the user slides open the notification shade and taps on our entry, the `Notification` is automatically canceled and goes away.

If we succeeded (the passed-in `Exception` is `null`), we further configure our `Notification` via more calls to the `Builder`:

- `setContentTitle()` and `setContentText()` supply the prose to display in the two lines of the notification shade entry for our `Notification`
- `setSmallIcon()` indicates the icon to display in the status bar or system bar when the `Notification` is active (in this case, specifying one supplied by Android itself)

- `setTicker()` supplies some text to be displayed in the status bar or system bar for a few seconds right when the `Notification` is displayed, so users who happen to be looking at their device at that time will get more information at a glance about what just happened that is demanding their attention (though only on Android 4.4 and older devices, as the ticker was discontinued with Android 5.0)

In addition, `setContentIntent()` supplies a `PendingIntent` to be invoked when the notification shade entry for our `Notification` is tapped. In our case, we create an `ACTION_VIEW` `Intent` for our file. To do this, we use `FileProvider.getUriForFile()`, as we are serving this PDF via a `FileProvider`. Hence, if the user taps on our notification shade entry, we will attempt to bring up a PDF viewer on the downloaded PDF file – whether this will succeed or not will depend upon whether there is a PDF viewer installed on the device.

If, instead, we did have an `Exception`, we use the same methods on `Builder` (minus `setContentIntent()`) to configure the `Notification`, but using different text and icons.

Then, we can call `notify()` on the `NotificationManager`, supplying our `Notification` (from `build()` on the `Builder`) and a locally-unique integer (`NOTIFY_ID`, defined as a static data member on the service). That integer can later be used with a `cancel()` method to remove the `Notification` from the screen, even if the user has not canceled it themselves (e.g., via tapping on it with `setAutoCancel(true)`).

Running this in a device or emulator will display the `Notification` upon completion of the download, with the download icon:

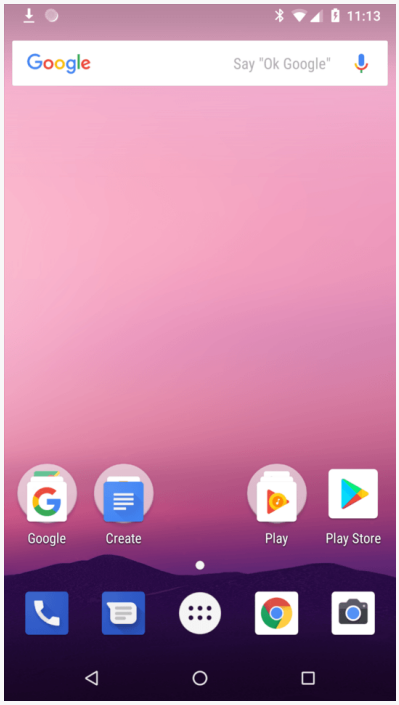


Figure 383: Sample Notification, on a Nexus 5X

Opening the notification shade displays our `Notification` details:

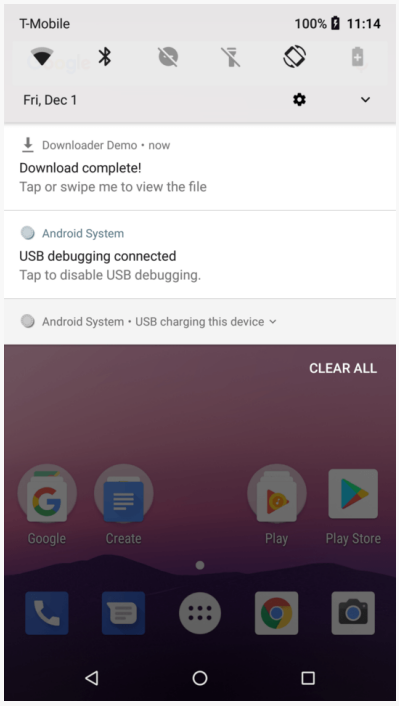


Figure 384: Sample Notification in Shade, on a Nexus 5X

Tapping on the shade entry will try to start a PDF viewer, perhaps bringing up a chooser if there are multiple such viewers on the device. Also, tapping on the shade entry will cancel the `Notification` and remove it from the screen.

The Activity-Or-Notification Scenario

Let us suppose that you are writing an email app. In addition to an "inbox" activity, you have an `IntentService`, scheduled via `AlarmManager`, to go check for new email messages every so often. This means, when your service discovers and downloads new messages, there are two possibilities:

- The user has your inbox activity in the foreground, and that activity should update to reflect the fact that there are new messages
- The user does *not* have your inbox activity in the foreground, so you want to display a `Notification` to alert the user of the new messages and lead them back to the inbox

However, ideally, the service neither knows nor cares whether the inbox activity is in the foreground, exists in the process but is not in the foreground, or does not exist

in the process (e.g., Android started a new process to handle this middle-of-the-night check for new email messages).

One way to handle this is via an event bus.

The recipe for the `Activity`-or-`Notification` pattern is:

1. Define an event (e.g., event class for greenrobot's EventBus, custom action string for `LocalBroadcastManager`)
2. Have your activity or fragment register to respond to these events while in the foreground (e.g., in `onResume()`) and unregister when leaving the foreground (e.g., `onPause()`). The activity or fragment can then update the UI in response to the event.
3. The service raises the event bus event at appropriate times.
4. By some means appropriate to the event bus implementation, the service needs to know whether an activity or fragment responded to the event, so it can raise a `Notification` if the event has not already been handled.

We will see three implementations of this pattern in the chapter on event bus alternatives.

Big (and Rich) Notifications

Android 4.1 introduced new `Notification` styles that automatically expand into a "big" area when they are the top `Notification` in the shade. These expanded `Notifications` can display more text (or a larger thumbnail of an image), plus add some action buttons to allow the user to directly perform more actions straight from the `Notification` itself.

And while these new `Notification` styles are only available on API Level 16 and higher, a familiar face has created a compatibility layer so our code can request the larger styles and still work on older devices.

The Styles

There are three main styles supplied for expanded `Notifications`. There is the `BigText` style:

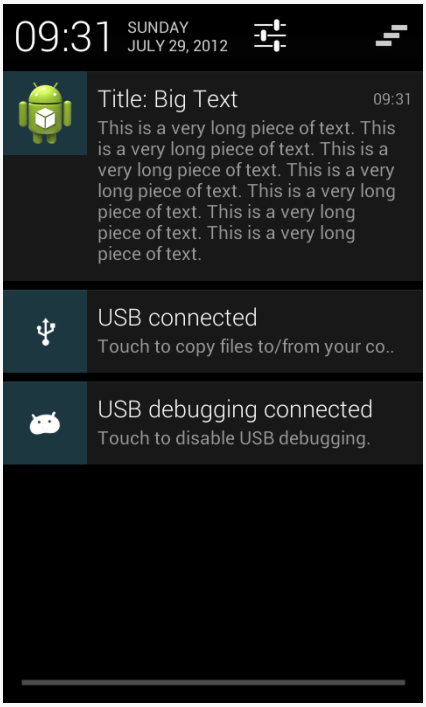


Figure 385: BigText Notification

We also have the `Inbox` style, which is the same basic concept but designed for several discrete lines of text:

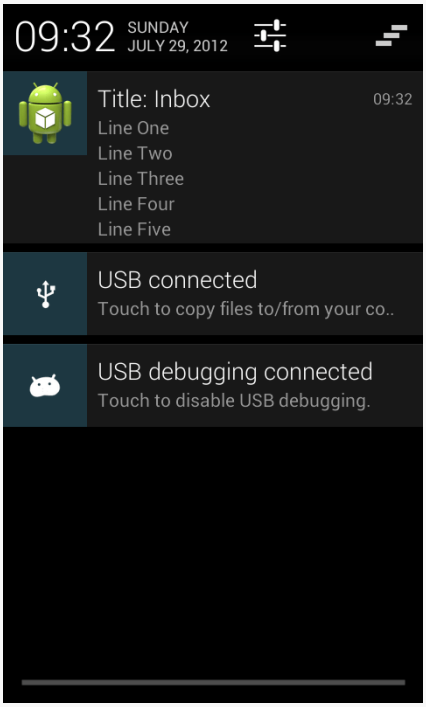


Figure 386: Inbox Notification

And, we have the `BigPicture` style, ideal for a photo, album cover, or the like:

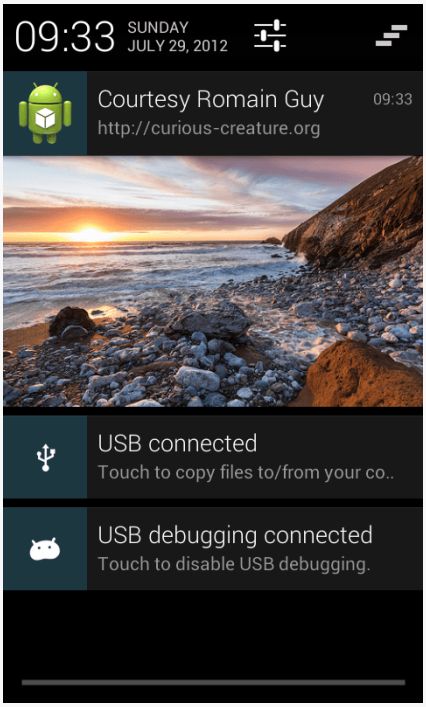


Figure 387: BigPicture Notification

(as noted in the screenshot, the photo is courtesy of Romain Guy, a former engineer on the core Android team and photography buff)

The Builders

`Notification.Builder` and `NotificationCompat.Builder` have been enhanced to support these new styles. Specifically:

- There is an `addAction()` method on the `Builder` class to define the action buttons, in terms of icon, caption, and `PendingIntent` that should be executed when the button is clicked
- There are style-specific builders, such as `Notification.InboxStyle`, that take a `Notification.Builder` and define the alternative expanded definition to be used when the `Notification` is at the top

The Sample

To see expanded notifications, take a peek at the `Notifications/BigNotify` sample application. This application consists of a single activity (`MainActivity`) that will raise a `Notification` and `finish()`, using `@style/Theme.Translucent.NoTitleBar`

to suppress the activity's own UI. Hence, the result of running the app is to display the `Notification` and do nothing else. While silly, it minimizes the amount of ancillary code involved in the project.

The process of displaying an expanded `Notification` is to first create the basic `Notification`, containing what you want to display for any non-expanded circumstance:

- Older devices that cannot display expanded `Notifications`, or
- Newer devices where the `Notification` is not the top-most entry in the notification shade, and therefore appears in the classic non-expanded form

Hence, in `onCreate()`, after getting our hands on a `NotificationManager`, we create a notification channel (if needed), then use `NotificationCompat.Builder` to create a regular `Notification`, wrapped in a private `buildNormal()` method:

```
private NotificationCompat.Builder buildNormal() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setAutoCancel(true)
   .setContentTitle(getString(R.string.download_complete))
   .setContentText(getString(R.string.fun))
   .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
   .setSmallIcon(android.R.drawable.stat_sys_download_done)
   .setPriority(Notification.PRIORITY_HIGH)
   .addAction(android.R.drawable.ic_media_play,
              getString(R.string.play),
              buildPendingIntent(Settings.ACTION_SETTINGS));

  return(b);
}
```

(from Notifications/BigNotify/app/src/main/java/com/commonsware/android/bignotify/MainActivity.java)

Most of what `buildNormal()` does is the same sort of stuff we saw with `NotificationCompat.Builder` earlier in this chapter. There are two things, though, that are new:

1. We call `setPriority()` to set the priority of the `Notification` to `PRIORITY_HIGH`. This means that this `Notification` *may* be displayed higher in the notification shade than it might ordinarily appear.
2. We call `addAction()` to add an action button to the `Notification`, to be shown in the expanded form. We are able to supply an icon, caption, and

PendingIntent, the latter created by a buildPendingIntent() method that wraps our desired Intent action string (here, Settings.ACTION_SETTINGS) in an Intent:

```
private PendingIntent buildPendingIntent(String action) {
  Intent i=new Intent(action);

  return(PendingIntent.getActivity(this, 0, i, 0));
}
```

(from Notifications/BigNotify/app/src/main/java/com/commonsware/android/bignotify/MainActivity.java)

Ordinarily, we might use this Builder directly, to raise the Notification we described. And, if we just wanted the action button to appear and nothing else new in the expanded form, we could do just that. But in our case, we also want to change the look of the expanded widget to a new style, InboxStyle. To do that, we need to wrap our Builder in a NotificationCompat.InboxStyle builder:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
  }

  NotificationCompat.Builder normal=buildNormal();
  NotificationCompat.InboxStyle big=
      new NotificationCompat.InboxStyle(normal);

  mgr.notify(NOTIFY_ID,
             big.setSummaryText(getString(R.string.summary))
                .addLine(getString(R.string.entry))
                .addLine(getString(R.string.another_entry))
                .addLine(getString(R.string.third_entry))
                .addLine(getString(R.string.yet_another_entry))
                .addLine(getString(R.string.low)).build());

  finish();
}
```

(from Notifications/BigNotify/app/src/main/java/com/commonsware/android/bignotify/MainActivity.java)

Each of these "big" builders has a set of methods that are unique to that type of builder to configure the look beyond what a standard `Notification` might have. Specifically, in this case, we call:

- `setSummaryText()`, to provide "the first line of text after the detail section in the big form of the template", in the words of the JavaDocs, though this does not necessarily mean what you think it does
- `addLine()`, to append several lines of text to appear in the `Notification`

It is the `Notification` created by our `NotificationCompat.InboxStyle` builder that we use with the call to `notify()` on `NotificationManager`.

The Results

If we run our app, we get this:

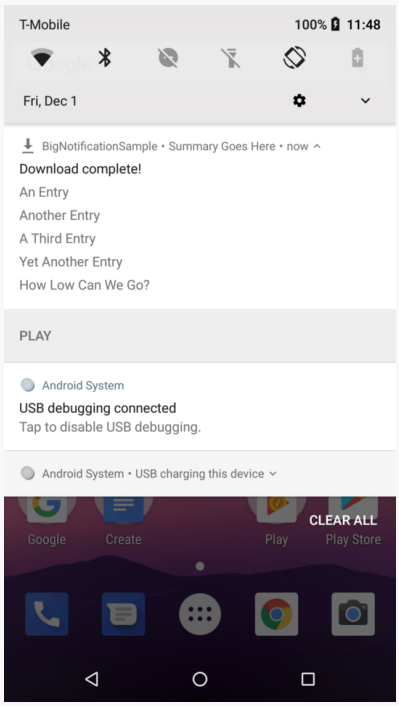


Figure 388: Expanded Notification in Shade, on Android 8.1

From top to bottom, we have:

- Our content text
- Our appended lines of text

- Our action button
- Our summary text

Note that this is the appearance when we are in expanded mode, at the top of the notification shade. If our `Notification` is not at the top, or if it is displayed on a pre-4.1 device, the appearance is the normal style, as defined by our `buildNormal()` method, though on Android 4.1+ devices the user can use a two-finger downward swipe gesture to expand the un-expanded `Notification`.

Foreground Services

If you have a service that will run for a substantial period of time, there is a risk that your process will still be terminated. That could be triggered by the user, or it could be the OS's own decision, based on the age of your process.

Generally speaking, this is a good thing for the user, because too many developers "leak" services, causing them to run unnecessarily, without adding value to the user, and tying up system RAM as a result.

But, what about services that *are* delivering value to the user for a long period? For example, what about a music player, where, in theory, the service is delivering value until the user presses some sort of "stop" button somewhere to turn off the music?

For those sorts of situations, you can flag a service as being a "foreground service".

Isn't "Foreground Service" an Oxymoron?

You might be forgiven for thinking that "foreground" and "service" are not designed to go together.

Partly, that is because we have overloaded the term "foreground".

A foreground service is *not* one that somehow takes over the screen. A foreground service *is* one that runs with foreground priority. That means:

- It will be treated similarly to the app that is in the UI foreground, from the standpoint of determining processes eligible for termination
- It will be classified as foreground from a CPU standpoint, rather than being relegated to the standard background process group

The former is what many developers want: a service (and process) that will not go away.

The latter is what many users fear: a service (and process) that is capable of stealing chunks of CPU time away from the game, video, or whatever else is truly in the foreground from a UI standpoint.

Services themselves, while useful, are best when used *sparingly*, only running when they are actively delivering value to the user. "This goes double" for foreground services.

Putting Your Service in the Foreground

Putting a service into the foreground is a matter of calling `startForeground()`. This method takes two parameters, the same two parameters that you would pass to `notify()` of `NotificationManager`:

- A prepared `Notification`
- A unique ID for that `Notification`

Android will then display the `Notification`. So long as the `Notification` is visible, your app's process will be given foreground priority.

You undo this by calling `stopForeground()`. `stopForeground()` takes a boolean parameter, indicating if the `Notification` should be removed (`true`) or not (`false`). Typically, you will pass `true`, so the `Notification` only clutters up the screen while you need it.

The `Notifications/Foreground` sample project is a clone of the `Notifications/DownloadNotify` sample that opened this chapter, adding in the use of `startForeground()` and `stopForeground()`.

Towards the top of `onHandleIntent()`, we call `startForeground()`, to *really* ensure that our process will remain intact long enough to complete the requested download:

```
    startForeground(FOREGROUND_ID,
      buildForegroundNotification(filename));
```

(from Notifications/Foreground/app/src/main/java/com/commonsware/android/foredown/Downloader.java)

This, in turn, uses a `buildForegroundNotification()` method to build the `Notification` that will be displayed while the service is categorized as being in the foreground:

```
private Notification buildForegroundNotification(String filename) {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setOngoing(true)
   .setContentTitle(getString(R.string.downloading))
   .setContentText(filename)
   .setSmallIcon(android.R.drawable.stat_sys_download);

  return(b.build());
}
```

(from Notifications/Foreground/app/src/main/java/com/commonsware/android/foredown/Downloader.java)

Note that we use `setOngoing(true)`, to indicate that this is an "ongoing" operation. This precludes the user from removing the `Notification` manually, as doing that would drop our process out of foreground priority.

At the end of `onHandleIntent()`, we call `stopForeground()` in a `finally` block, to ensure that it gets called:

```
  try {
    // rest of code omitted for brevity

    raiseNotification(i, output, null);
  }
  catch (IOException e2) {
    raiseNotification(i, null, e2);
  }
  finally {
    stopForeground(true);
  }
```

We pass `true` to `stopForeground()` to remove the `Notification`. From the user's perspective, we could just as easily have passed `false`, as the `Notification` used with `startForeground()` will also be removed once our service is destroyed, which will happen shortly after `onHandleIntent()` ends.

If you want to update the foreground `Notification`, you can either:

- Call `notify()` again with the same notification ID and a fresh `Notification`, as you would use to update any `Notification`, or
- Simply call `startForeground()` again, with the same notification ID and a fresh `Notification`

We will see this particular practice in use later in the book, where we use a foreground service's `Notification` to control recording a screencast of an Android device.

The Malformed Notification

Of course, some developers do not play nicely with the other kids.

A technique that had been around for a while was for an app to pass an intentionally-flawed `Notification` to `startForeground()`. While Android would blow up silently somewhere internally actually trying to display the `Notification`, the foreground status was still granted. This resulted in behavior reminiscent of the long-since-deprecated `setForeground()` method.

`setForeground()` allowed a service to get foreground priority with no repercussions. Not surprisingly, lots of developers used it, as they decided that their app was more important than any other apps on the device. `setForeground()` was replaced by `startForeground()`, adding in the `Notification` requirement, to put a "cost" on foreground status. The malformed-`Notification` trick allowed developers to avoid that cost.

In Android 4.3, if you pass a malformed `Notification` to `startForeground()`, Android will create one for you, featuring your app's launcher icon, and use it instead. Hence, on Android 4.3 and higher, you cannot hide your foreground status from the user.

Disabled Notifications

Because apps have the ability to display larger-than-normal `Notifications`, plus force them towards the top of the list via priority levels, Android has given users the ability to disable `Notifications` on a per-app basis. The degree of control, and the way the user sets up that control, depends upon Android version.

Android 4.x

Users visiting an app's page in Settings will see a "Show notifications" checkbox:

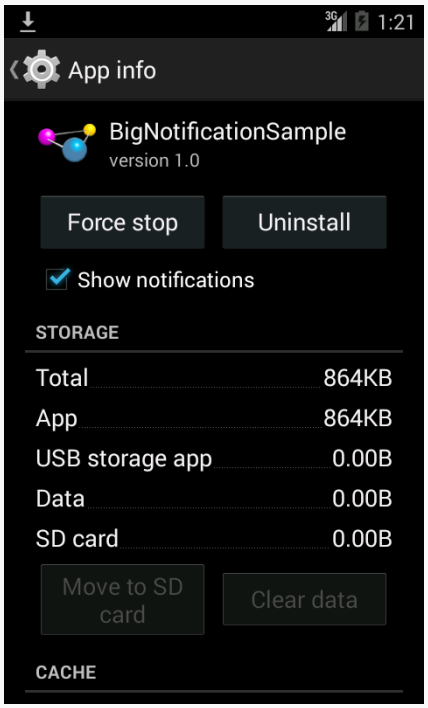


Figure 389: Show Notifications Checkbox, on Android 4.4

If the user unchecks the checkbox and agrees on the resulting confirmation dialog, your requests to raise a `Notification` will be largely ignored. An error message will appear in Logcat ("Suppressing notification from package ... by user request"), but no exception will be raised. Further, there does not appear to be an API for you to determine if the notification will actually be displayed.

Also note that, on Android 4.2+, if the user blocks notifications, it also blocks `Toast` requests from your app.

And, also note that this setting survives an uninstall of your app. If the user unchecks this checkbox, uninstalls your app, then reinstalls your app, the checkbox is still unchecked, meaning that notifications will still be blocked.

The one notable exception to this blocking, as of Android 4.3, is that the `Notification` associated with a foreground service will not be blocked. It will always appear, even if the user unchecked "Show notifications" for your app in Settings.

Android 5.0+

In the "Sound & notification" area of Settings, the user can tap on an "App notifications" option, and from there choose an app. This brings up a screen where the user can "block" (i.e., disable) notifications:

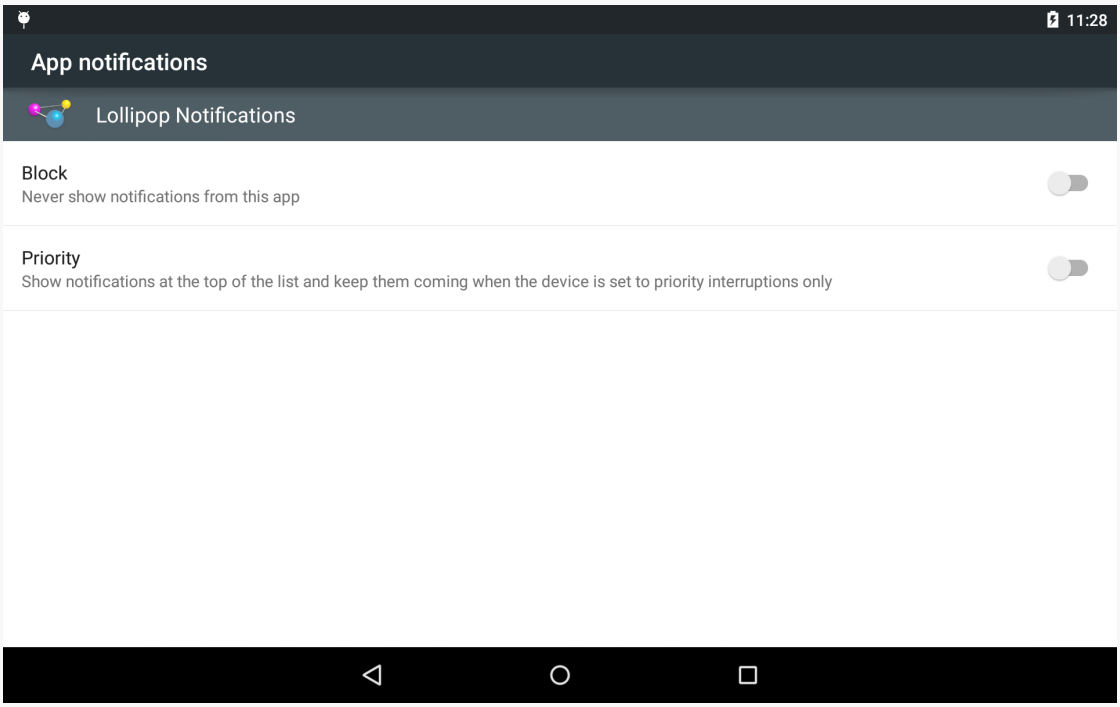


Figure 390: "App notifications" in Settings, on Android 5.0

The top "Block" `SwitchPreference`, if toggled on, will prevent app notifications from being displayed.

The bottom "Priority" `SwitchPreference`, if toggled on, marks this app's notifications as being "priority". Then, in the main "Sound & notification" area of Settings, the user can tap on an "Interruptions" option:

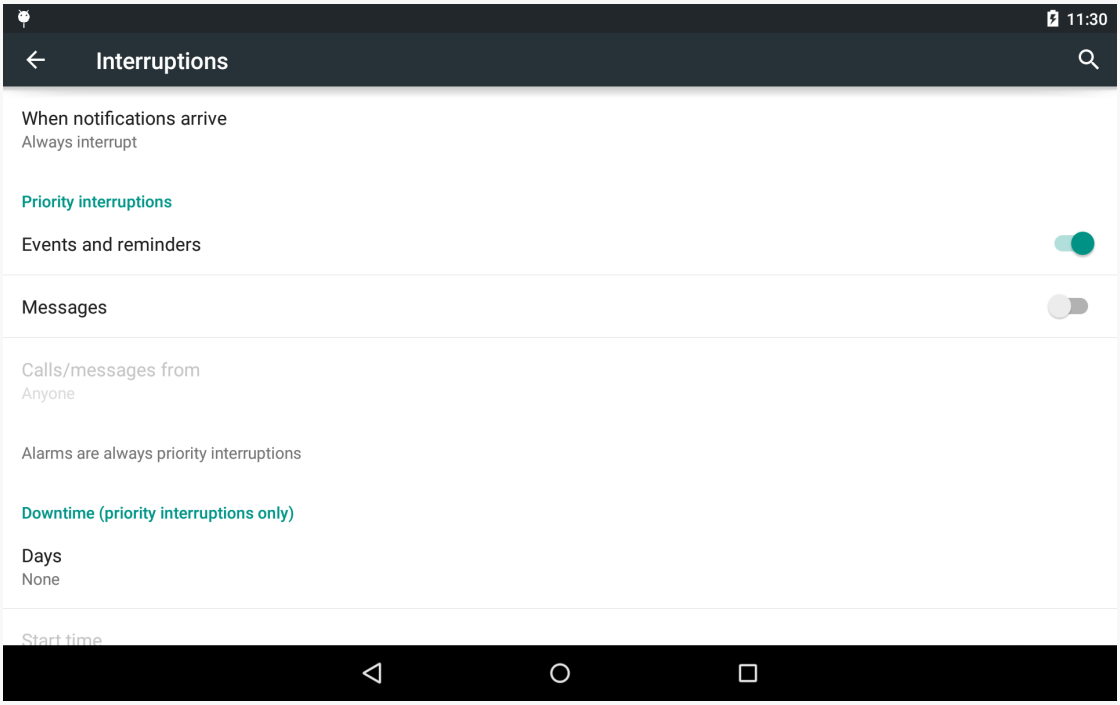


Figure 391: "Interruptions" in Settings, on Android 5.0

If the user toggles the "When notifications arrive" option to "Allow only priority interruptions", then those apps that the user configures as "Priority" in "App notifications" will behave normally. Other apps' notifications will appear, but will not play a ringtone or vibrate the device.

If the user toggles the "When notifications arrive" option to "Don't interrupt", all notifications — even those marked as "Priority" — will have their ringtones and vibrations suppressed.

Android 8.0+

Users have similar control capability on Android 8.0 and higher as they did on Android 5.0 through 7.1. Now, however, they are divided into channels, as will be explained in the chapter on advanced notifications.

Advanced Notifications

Notifications are those icons that appear in the status bar (or system bar on tablets), typically to alert the user of something that is going on in the background or has completed in the background. Many apps use them, to let the user know of new email messages, calendar reminders, and so on. Foreground services, such as music players, also use notifications, to tell the OS that they are part of the foreground user experience and to let the user rapidly return to the apps to turn the music off.

There are other tricks available with the `Notification` object beyond those originally discussed in an earlier chapter.

Prerequisites

Understanding this chapter requires that you have read the chapter on basic notifications and the section on `RemoteViews` in the chapter on basic app widgets.

Being a Good Citizen

Users have a love/hate relationship with apps that use notifications:

- They love apps that raise notifications for events that the user cares about...
- ...but they hate apps that raise notifications for events that the user does not care about (e.g., Evernote's "please confirm your email" notifications)
- They love apps that provide control over when and how notifications appear...
- ...but they hate apps that display notifications solely because the developer wanted them (e.g., ads in notifications)

- They love apps that use notifications to let the user control some background operation, like media playback...
- ...but they hate apps that have ongoing notifications for no obvious reason (e.g., developers trying to use a foreground service to keep their process around, rather than using `AlarmManager`, `JobScheduler`, or other means of doing work periodically)
- They love apps that set up notifications for use in different scenarios, such as supporting Android Wear devices...
- ...but they hate apps that wind up flooding their wrist (or eyes, or other wearable locations) with notifications that have to be individually dismissed

And so on.

Users' discomfort with how apps handle notifications is why Android allows users to disable notifications.

Some of the items in this chapter, particularly those surrounding Android Wear, can help you improve user satisfaction with your notification strategy and tactics. Yet, at the same time, misuse of notifications is magnified by Wear, as Wear takes extra steps to get the user to pay attention to the notifications, with possibly disastrous results for your Play Store reviews.

In short, your objective with notifications is to be a good citizen:

- Have a reasonable default mode for your notifications
- Allow users to tailor that mode to better suit their needs, where practical

More About Channels

Android 8.0 moved a lot of notification configuration out of `NotificationCompat.Builder` and into a new `NotificationChannel` class, and moving their effects from being per-notification to per-channel. When you configure a channel, all notifications in that channel will exhibit the behavior defined for the channel.

What You Do with Channels

Many configuration options formerly set on a `Notification` are now set on a `NotificationChannel`:

You Used to Call This on `Notification`	You Now Call This on `NotificationChannel`
`setDefaults()`	no real analogue
`setLights()`	`enableLights()` and `setLightColor()`
`setPriority()`	`setImportance()`
`setSound()`	`setSound()`
`setVibration()`	`enableVibration()` and `setVibrationPattern()`
`setVisibility()`	`setLockscreenVisibility()`

The "enable" methods (`enableLights()`, `enableVibration()`) opt you into whatever the device default behaviors are for those features.

As a result of these things moving to `NotificationChannel`, the user can control the behavior. You might *request* to enable vibration, but the user will be able to override your choice, for example.

In addition to these options, a `NotificationChannel` also configures:

- Whether this channel's notifications should bypass do-not-disturb settings (`setBypassDnd()`)
- Whether a launcher icon badge should be used with this channel's notifications (`setShowBadge()`)

Channels can also be associated into channel groups. This is purely for organizational purposes; groups show up on the screen that the user sees to help cluster related channels together. Beyond that, channels in groups behave identically to channels not in groups.

Creating Channel Groups

A `NotificationChannelGroup` is the Java class embodiment of a channel group. A `NotificationChannelGroup` consists purely of a `String` unique identifier and a `CharSequence` (e.g., `String`) display name to show the user. In most cases, you will want the display name to come from a string resource, for translation purposes.

You need to register your channel groups with Android. To do this, you will need to call one of the following methods on `NotificationManager`:

- `createNotificationChannelGroup()`, to create a single channel group

- `createNotificationChannelGroups()`, to create a `List` of channel groups

The Notifications/Channels sample project demonstrates the use of notification channels and channel groups. The `MainActivity` has an `initGroups()` method that defines two channel groups:

```
private void initGroups() {
  ArrayList<NotificationChannelGroup> groups=new ArrayList<>();

  groups.add(new NotificationChannelGroup(GROUP_UPDATES,
    getString(R.string.group_name_updates)));
  groups.add(new NotificationChannelGroup(GROUP_PROMO,
    getString(R.string.group_name_promo)));

  mgr.createNotificationChannelGroups(groups);
}
```

(from Notifications/Channels/app/src/main/java/com/commonsware/android/notify/channel/MainActivity.java)

Here, the display names come from string resources, and the channel group IDs are string constants defined on `MainActivity` itself.

Creating Channels

Creating channels, in turn, involves configuring `NotificationChannel` objects and calling `createNotificationChannel()` on `NotificationManager` for each channel.

`MainActivity` has three methods for defining three separate channels, in those two channel groups:

```
private void initContentChannel() {
  NotificationChannel channel=
    new NotificationChannel(CHANNEL_CONTENT,
      getString(R.string.channel_name_content),
      NotificationManager.IMPORTANCE_LOW);

  channel.setGroup(GROUP_UPDATES);
  mgr.createNotificationChannel(channel);
}

private void initBattleChannel() {
  NotificationChannel channel=
    new NotificationChannel(CHANNEL_BATTLE,
      getString(R.string.channel_name_battle),
      NotificationManager.IMPORTANCE_HIGH);
```

```
  channel.setGroup(GROUP_UPDATES);
  channel.setShowBadge(true);
  mgr.createNotificationChannel(channel);
}

private void initCoinsChannel() {
  NotificationChannel channel=
    new NotificationChannel(CHANNEL_COINS,
      getString(R.string.channel_name_coins),
      NotificationManager.IMPORTANCE_DEFAULT);

  channel.setGroup(GROUP_PROMO);
  mgr.createNotificationChannel(channel);
}
```

(from Notifications/Channels/app/src/main/java/com/commonsware/android/notify/channel/MainActivity.java)

The constructor takes a unique `String` identifier for the channel, a user-facing display name for the channel, and the importance of the notifications in that channel. Again, typically you will want to use a string resource for the display name, as the sample code demonstrates.

Between creating the `NotificationChannel` and registering it with `createNotificationChannel()` on `NotificationManager`, you can call various setter methods to configure the way that this channel behaves, beyond the importance. For example, if you are using channel groups, you will need to call `setGroup()` on the `NotificationChannel`, passing in the channel group ID.

All of these methods are invoked from `onCreate()` of `MainActivity`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  mgr=getSystemService(NotificationManager.class);

  if (mgr.getNotificationChannel(CHANNEL_CONTENT)==null) {
    initGroups();
    initContentChannel();
    initBattleChannel();
    initCoinsChannel();
  }

  setContentView(R.layout.activity_main);
}
```

(from Notifications/Channels/app/src/main/java/com/commonsware/android/notify/channel/MainActivity.java)

In particular, we only call those methods if we do not already have the `CHANNEL_CONTENT` channel defined, as determined via a call to `getNotificationChannel()` on `NotificationManager`. This `if` check serves two purposes:

1. We avoid running through a bunch of initialization code for things that we have already initialized
2. We cannot change notification channels or channel groups once we create them, as we will examine more later in this chapter

What the User Sees

When the user visits the notification settings for this app, all three channels appear, clustered into the two groups:

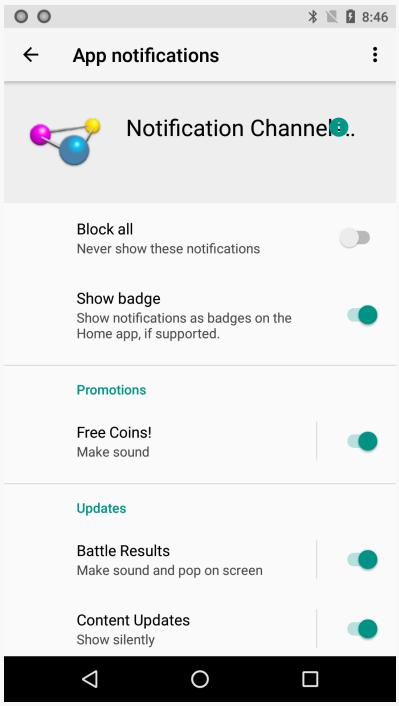


Figure 392: Notification Settings, Showing Three Channels in Two Groups

The groups appear to be sorted alphabetically, not in the order that they are defined.

Tapping on the channel itself brings up a screen for configuring the details of that channel:

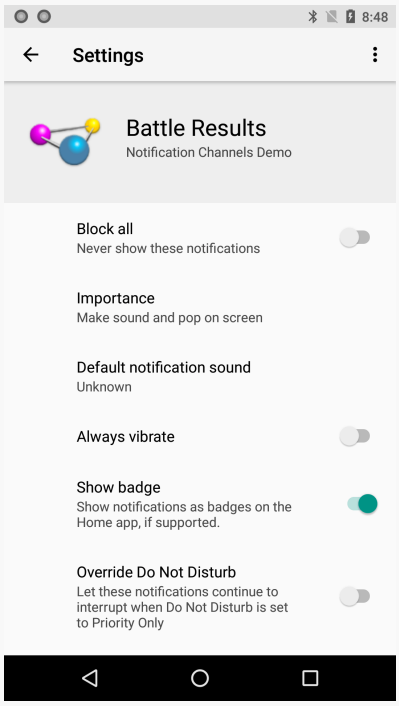


Figure 393: Notification Settings, Showing Channel Configuration

Of note, if the importance calls for a sound, and you did not supply a `Uri` to some ringtone via `setSound()` on the `NotificationChannel`, the user will see that the "Default notification sound" is "Unknown", though this really means that the default notification sound will be used.

You can bring up the activity for configuring channel settings yourself, via an `ACTION_CHANNEL_NOTIFICATION_SETTINGS` `Intent` and a call to `startActivity()`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.settings) {
    Intent i=new Intent(Settings.ACTION_CHANNEL_NOTIFICATION_SETTINGS);

    i.putExtra(Settings.EXTRA_CHANNEL_ID, CHANNEL_BATTLE);
    i.putExtra(Settings.EXTRA_APP_PACKAGE, getPackageName());
    startActivity(i);
  }
```

```
    return super.onOptionsItemSelected(item);
  }
```

(from Notifications/Channels/app/src/main/java/com/commonsware/android/notify/channel/MainActivity.java)

This requires two extras:

- `EXTRA_APP_PACKAGE` is your application ID, returned via `getPackageName()` or `BuildConfig.APPLICATION_ID`
- `EXTRA_CHANNEL_ID` is the channel ID of the channel whose settings you wish to display

Once and Done

Once you create a notification channel, control over the channel settings resides with the user. You cannot modify the settings of that channel. Consider your settings to be defaults; the user is welcome to modify the channel from those defaults as the user sees fit.

To help prevent developers from deleting and re-creating channels, while you can call `deleteNotificationChannel()` on `NotificationManager`, that channel will still be visible to the user ("Deleted channels remain visible in notification settings, as a spam prevention mechanism.")

Backwards Compatibility

There is no backport of notification channels.

`NotificationCompat.Builder` takes a channel ID, but you will need to use the native `NotificationChannel` class to define the channel. And since that class only exists on Android 8.0+, unless your `minSdkVersion` is 26 or higher, you will need to make sure that you use `NotificationChannel` only on suitable devices, using the recipe that we saw in the original chapter on notifications:

```
if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
  mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
  mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
    "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
}
```

Wear? There!

The humble `Notification` has been steadily advancing over the past few years, with "big" styles and the like adding new capabilities for newer devices.

Android Wear takes notifications to a new level, by having the notification not only appear on the user's device, but also on wearables connected to that device.

The good news is that this works "out of the box". There is nothing you absolutely need to do in your app to get your notifications to appear on a Wear device.

The bad news is that the "out of the box" experience may be poor, as a `Notification` approach that is fine for devices that reside in pockets and backpacks might be inappropriate for wrists and eyes.

With that in mind, let's see what some notification samples from earlier in the book behave like when they are run on a phone connected to a Wear device.

NOTE: For this section, and the rest of this chapter, "primary device" will refer to the user's phone or tablet that the "Wear device" will be connected to.

Simple Notification

The `Notifications/DownloadNotify` sample project allows the user to download a PDF file, raising a `Notification` when that download is complete.

With a Wear device paired with the phone, the `Notification` also appears on the device, first as a "mini card":

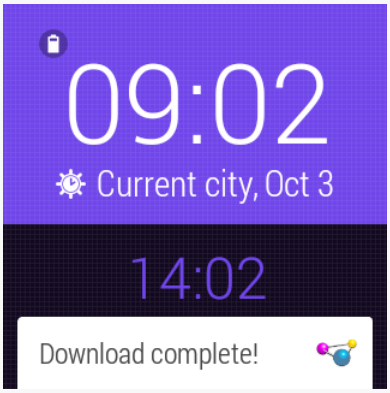


Figure 394: Simple Notification on Wear, As Originally Displayed, On Samsung Galaxy Gear

Swiping up on that will bring up the full card:

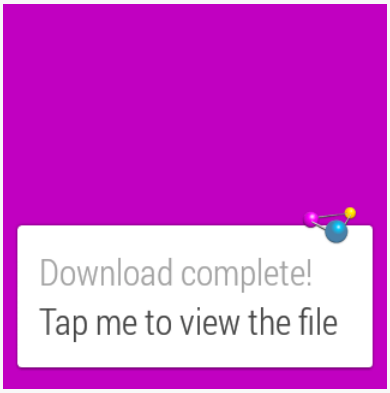


Figure 395: Simple Notification on Wear, Full, On Samsung Galaxy Gear

Swiping to the right will bring up the action associated with `setContentIntent()` on the `NotificationCompat.Builder`:

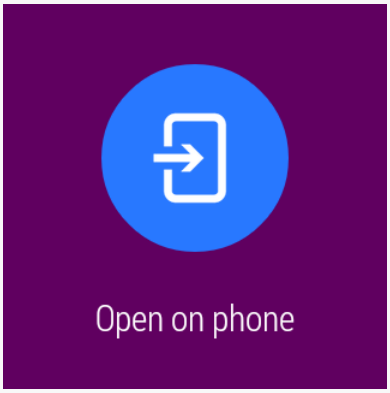


Figure 396: Simple Notification on Wear, Default Action, On Samsung Galaxy Gear

Tapping on that dismisses the `Notification` on the Wear device and the primary device, plus it invokes the `PendingIntent` on the phone itself (in this case, opening up the PDF file).

This is a fine example of a `Notification` that perhaps should *not* appear on the Wear device. The fact that the download completed is interesting but not all that important. Furthermore, the user cannot do anything about this download other than to pull out the primary device to see the PDF. Low-priority primary-device-centric notifications generally should be shown on the primary device alone, not on the Wear device. We will see how to do that later in this chapter.

"Big" Style and Action Button

The `Notifications/BigNotify` sample application wrapped a regular `Notification` in a `NotificationCompat.InboxStyle` "big" `Notification`, one with both a regular action and a separate "Play" action button.

As before, with a Wear device paired with the phone, the `Notification` also appears on the device, first as a "mini card":

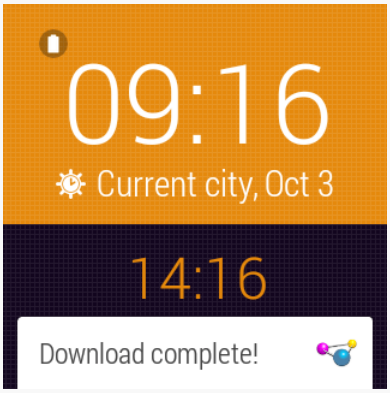


Figure 397: Big Notification on Wear, As Originally Displayed, On Samsung Galaxy Gear

However, this time, when the user swipes up to show the full card, it is the `InboxStyle` version that appears, albeit without the summary text:

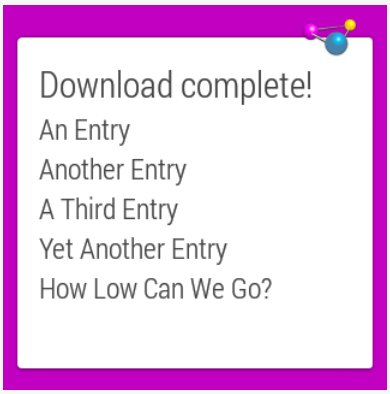


Figure 398: Big Notification on Wear, Full, On Samsung Galaxy Gear

Swiping to the right shows our actions, starting with the custom “Play” action:

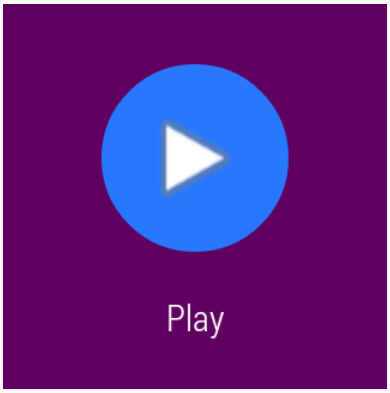


Figure 399: Big Notification on Wear, Play Action, On Samsung Galaxy Gear

...followed by the default action:

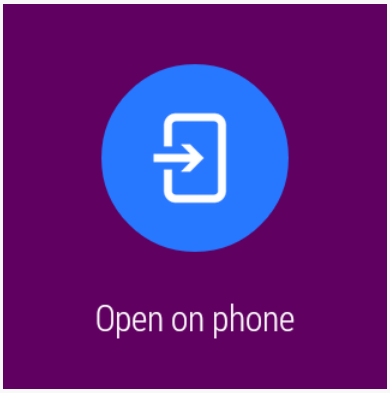


Figure 400: Big Notification on Wear, Default Action, On Samsung Galaxy Gear

Tapping on either action will cause the primary device to invoke its `PendingIntent`, but only the default action dismisses the `Notification` from both devices. The custom "Play" action does not.

Foreground Service

The `Notifications/Foreground` sample project is another version of the download-the-file sample, but this time uses a `Notification` and `startForeground()` to mark the service as a foreground service while it is downloading things.

This particular sample does not spend much time in the foreground state, so for testing purposes, you may want to add a `SystemClock.sleep()` call to the service,

between the `startForeground()` and `stopForeground()` calls, to better examine the behavior while the foreground service `Notification` is around.

However, in truth, that modification is probably not necessary... as the foreground service `Notification` is not displayed on the Wear device, only on the primary device. This is by design. The expectation is that you would use a Wear app to control your service from the Wear device, not some un-dismissable card.

Stacking Notifications

If you are writing an email client, and you want to use a `Notification` to let the user know about new email messages, you do *not* want to raise a separate `Notification` for each email. Users will come to your home with pitchforks and torches... and not to help you with farming.

Instead, the vision is that you update an existing `Notification` with new content. For example, you might start with a regular `Notification` for the first received email. Then, when the second one comes in, you replace that `Notification` with one that has a simple summary ("2 messages are in your inbox!"), plus perhaps an `InboxStyle` "big" `Notification` variant that could show the subject lines for both of those messages.

Android Wear devices, however, add an interesting wrinkle: you want the `Notification` to be informative about the event itself. You want the user to be able to make an informed decision about whether they should pull out their primary device to read the new messages, and that decision is only partly based on how many messages there are. Users will want to know more about the outstanding messages (sender and/or subject line) to help them make that decision... at least to a point. If there are 57 unread messages, users may get frustrated dealing with all of those as individual items on the wearable itself.

The pattern here, then, takes advantage of some "group" capabilities added to `NotificationCompat`:

- Raise one "summary" `Notification`, that will only be shown on the primary device, with the same sort of "2 messages are in your inbox!" information that you would have used without considering Wear
- Raise individual notifications for individual messages that will appear on the Wear device

- Collect all of those in a "group", so the primary device shows only the summary and the Wear device shows only the individual ones

This can be seen in action in the `Notifications/Stacked` sample project.

The setup is reminiscent of the "big" style one from the original chapter on `Notification`. However, this time, there are a total of three `Notification` objects created: two for individual events for the Wear device, and one summary one for the primary device.

However, to make this work, we need a new version of the `support-v13` library from the Android Support package: `20.0.0` (or higher), as it is where the extra compatibility smarts were added to support this whole group-and-summary construct. Hence, in `build.gradle`, we have `implementation 'com.android.support:support-v13:20.0.0'`.

Similarly, while we will still use `NotificationCompat` for creating the `Notification` objects, we will *not* use `NotificationManager` for displaying them. Instead, we need to use `NotificationManagerCompat` from the Android Support package. While the `NotificationManager` API has not changed to support the group-and-summary pattern, the *implementation* has, and `NotificationManagerCompat` gives us a version of that implementation that can work on compatible devices and gracefully degrade on older ones. However, since the API did not change, it is easy to miss this requirement, use `NotificationManager`, and not quite get the desired results. Notably, the primary device will wind up showing all three notifications, not just the summary as we want.

Hence, our `MainActivity` will hold onto a `NotificationManagerCompat` as a data member, initialized in `onCreate()`:

```
private NotificationManagerCompat mgrCompat=null;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  NotificationManager mgr=
    (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
```

```
    }

    mgrCompat=NotificationManagerCompat.from(this);

    showWearOne();
    showWearTwo();
    showSummary();

    finish();
  }
```

(from Notifications/Stacked/app/src/main/java/com/commonsware/android/stacked/MainActivity.java)

The three `show...()` methods are each responsible for raising one `Notification`: `showWearOne()` and `showWearTwo()` are ones that will wind up on the Wear device, and `showSummary()` will show the summary `Notification` for use on the primary device.

Beyond using `NotificationManagerCompat` instead of `NotificationManager`, the only substantial difference is the use of `setGroup()` and `setGroupSummary()` methods on the `NotificationCompat.Builder`.

`setGroup()` associates the `Notification` with a group, identified by a `String` key. On a Wear device, notifications that are part of a group will be shown stacked as part of a single card by default. So, the two `showWear...()` methods call `setGroup()` as part of building the `Notification`:

```
private void showWearOne() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setAutoCancel(true)
      .setDefaults(Notification.DEFAULT_ALL)
      .setContentTitle(getString(R.string.entry))
      .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
      .setSmallIcon(android.R.drawable.stat_sys_download_done)
      .setGroup(GROUP_SAMPLE);

  mgrCompat.notify(NOTIFY_ID2, b.build());
}

private void showWearTwo() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setAutoCancel(true)
      .setDefaults(Notification.DEFAULT_ALL)
      .setContentTitle(getString(R.string.another_entry))
      .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
      .setSmallIcon(android.R.drawable.stat_sys_download_done)
      .setGroup(GROUP_SAMPLE);
```

```
    mgrCompat.notify(NOTIFY_ID3, b.build());
  }
```

(from Notifications/Stacked/app/src/main/java/com/commonsware/android/stacked/MainActivity.java)

`setGroupSummary()` indicates a particular `Notification` that should serve as the summary for its group. This `Notification` will not be passed to the Wear device, and it replaces all other notifications for this group on the primary device. Hence, `showSummary()` (or, more accurately, the `buildNormal()` method that creates the base `Notification` for the summary) uses `setGroupSummary()`:

```
  private void showSummary() {
    NotificationCompat.Builder normal=buildNormal();
    NotificationCompat.InboxStyle big=
        new NotificationCompat.InboxStyle();

    big.setSummaryText(getString(R.string.summary))
        .addLine(getString(R.string.entry))
        .addLine(getString(R.string.another_entry));

    mgrCompat.notify(NOTIFY_ID, normal.setStyle(big).build());
  }

  private NotificationCompat.Builder buildNormal() {
    NotificationCompat.Builder b=
      new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setContentTitle(getString(R.string.download_complete))
        .setContentText(getString(R.string.fun))
        .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .setGroup(GROUP_SAMPLE)
        .setGroupSummary(true);

    return(b);
  }
```

(from Notifications/Stacked/app/src/main/java/com/commonsware/android/stacked/MainActivity.java)

Note that you need to use `setGroupSummary()` on a `NotificationCompat.Builder` on which you have also called `setGroup()`, to identify the group for which this `Notification` is a summary.

When you run this, the primary device shows the summary `Notification`:

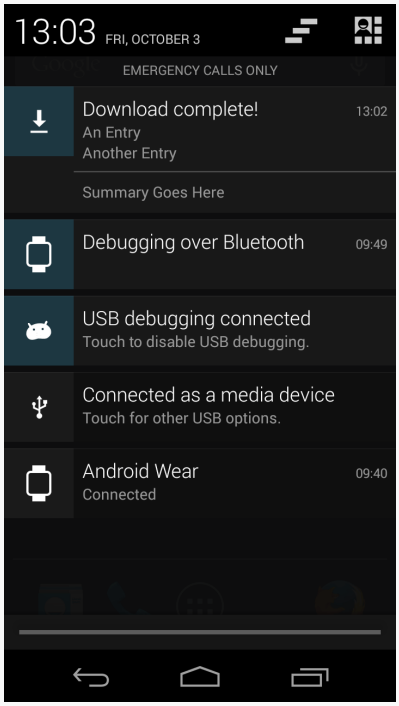


Figure 401: Stacked Notifications, Summary on Primary Device

On the Wear device, you will see the two original notifications as part of a single card at the outset:

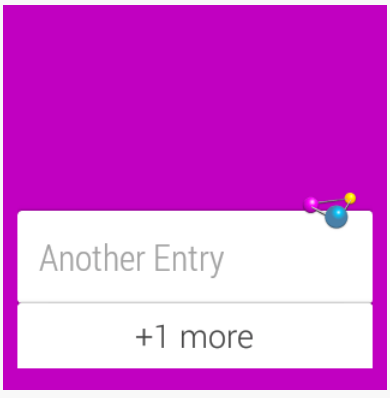


Figure 402: Stacked Notifications, Stacked on Wear Device

Tapping on the stack brings up separate mini cards for each individual `Notification`:

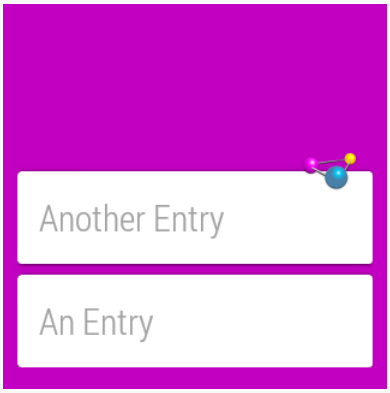


Figure 403: Stacked Notifications, Expanded Stack on Wear Device

…And the Passage of Time

Of course, this sample is artificially simple, like most of the samples in this book.

In the sample, we are raising all three notifications all at once. That is certainly conceivable, but it is not especially likely. A more likely scenario is that the mix of notifications needs to change over time, based upon continuing events, such as a trickle of new unread email messages for an email client.

This adds a few complexities to what you need to implement all of this properly.

The big thing is that your persistent data model (e.g., database) needs to have enough information for you to know how to notify the user about the next event, when that event occurs. Using the email client as an example:

- We start off in the "steady state" of no unread email messages and, therefore, no notifications from our app.
- A new email message arrives. At this point, we want to show a regular `Notification` on both the Wear device and the primary device, with the sender and subject line of the unread message.
- A second new email message arrives later. At this point, we want to show another regular `Notification` (requiring a separate notification ID) for the Wear device, but also show a summary `Notification` for the primary device. For all that to work, we need to know this *is* a second unread message, and that the user has not read the first message in between the two incoming messages. And, we need to know enough details about the unread messages to format the summary properly.

This gets even more complex when events "stack themselves" (e.g., one poll of the mail server results in two unread messages), in addition to having to deal with user input (e.g., user clears the notification stack from either device, yet does not read the messages).

Among other things, you cannot rely upon static data members as being the sole source of your `Notification`-related data, as your process may be terminated in between events. You are welcome to use it as a cache, in case your process does happen to survive long enough to process more than one event, but you will need to also save this data to a persistent store, so that you can properly handle new events requiring `Notification` changes with your process having been terminated since the last `Notification`-related event.

Avoiding Wear

Sometimes, you will want to raise a `Notification` that does not make sense to show on a Wear device, only on the primary device. In the case of the group summary for the stacked notifications, this primary-only behavior happens automatically. In other cases, though, you will need to call `setLocalOnly()` on the `NotificationCompat.Builder` to tell the framework that this `Notification` should only be displayed on the current device.

The Notifications/BigLocal sample project demonstrates this, through a clone of the `Notifications/BigNotify` sample that calls `setLocalOnly(true)` as part of configuring the `Notification`:

```
private NotificationCompat.Builder buildNormal() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setAutoCancel(true)
   .setDefaults(Notification.DEFAULT_ALL)
   .setContentTitle(getString(R.string.download_complete))
   .setContentText(getString(R.string.fun))
   .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
   .setSmallIcon(android.R.drawable.stat_sys_download_done)
   .setTicker(getString(R.string.download_complete))
   .setPriority(NotificationCompat.PRIORITY_DEFAULT)
   .setLocalOnly(true)
   .addAction(android.R.drawable.ic_media_play,
       getString(R.string.play),
       buildPendingIntent(Settings.ACTION_SETTINGS));
```

```
    return(b);
  }
```

(from Notifications/BigLocal/app/src/main/java/com/commonsware/android/biglocal/MainActivity.java)

Note that we do not need to use `NotificationManagerCompat` for local-only behavior — simply calling `setLocalOnly(true)` on an up-to-date `NotificationCompat.Builder` will suffice.

Running this sample provides the same behavior as `Notifications/BigNotify`, except that the `Notification` only appears on the primary device, not the Wear device.

Other Wear-Specific Notification Options

Configuring stacked notifications, and opting into local-only behavior when needed, should give you Wear behavior that is acceptable. Right now, Android Wear is fairly nascent, and therefore it may not behoove you to do much more than this, as you decide how to prioritize your engineering time.

However, there are other things that you can do to further tailor your notifications on Wear that can improve user satisfaction, if you wish for Wear to be a key part of your marketing message.

Pages

On the primary device, the amount of information you can provide in a `Notification` is intentionally capped. This prevents a `Notification` from drowning out its peers. The cap is not a big problem, simply because the whole UI for the app raising the `Notification` is usually just a tap away.

With a Wear device, though, the whole UI for the app raising the `Notification` involves pulling out the primary device.

Hence, it might be nice to provide some additional information to the Wear user, so that perhaps they can make a more informed decision as to whether it is worthwhile to open up their primary device. In Wear terms, this involves adding more “pages” to a `Notification`.

To do this, you must:

- Create the second (and additional) pages as their own separate `Notification` objects, probably via a `NotificationCompat.Builder`
- Use a `NotificationCompat.WearableExtender` to teach the primary `Notification` about the additional pages
- Raise the primary `Notification` using a `NotificationManagerCompat` variant of the system service

We can see this in action in the [Notifications/Pages] sample project. This is a clone of `Notifications/BigNotify`, where we make the "big" content be on a second page.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  NotificationManager mgr=
    (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
  }

  NotificationManagerCompat mgrCompat=
      NotificationManagerCompat.from(this);
  NotificationCompat.Builder normal=buildNormal();
  NotificationCompat.InboxStyle big=
      new NotificationCompat.InboxStyle();

  big.setSummaryText(getString(R.string.summary))
      .addLine(getString(R.string.entry))
      .addLine(getString(R.string.another_entry))
      .addLine(getString(R.string.third_entry))
      .addLine(getString(R.string.yet_another_entry))
      .addLine(getString(R.string.low));

  NotificationCompat.Builder bigPage=
      new NotificationCompat.Builder(this, CHANNEL_WHATEVER)
          .setStyle(big);
  NotificationCompat.Builder twoPages=
      new NotificationCompat.WearableExtender()
          .addPage(bigPage.build())
          .extend(normal);

  mgrCompat.notify(NOTIFY_ID, twoPages.build());
```

```
    finish();
  }
```

(from Notifications/Pages/app/src/main/java/com/commonsware/android/pages/MainActivity.java)

Here, we:

- Create a `NotificationManagerCompat` instance
- Create the primary ("normal") `Notification`, using the same process as before
- Create the `InboxStyle` structure with our expanded content
- Wrap that "big" style in another `Notification` via a `NotificationCompat.Builder`, using the `setStyle()` method to associate the "big" style with the `Notification`
- Create a `NotificationCompat.WearableExtender`, tell it to add the second page using `addPage()`, and tell it to apply that second page to the primary `Notification` via the `extend()` method
- Use `notify()` as normal to raise the `Notification`, using the already-created `NotificationManagerCompat` instance

On the primary device, we just see the primary `Notification` content:

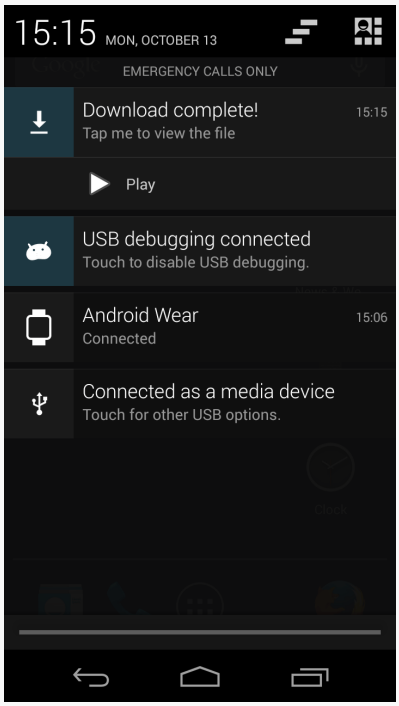


Figure 404: Pages Demo, on a Galaxy Nexus

On the Wear device, we see the main `Notification` and the second page as separate pages on the wearable:

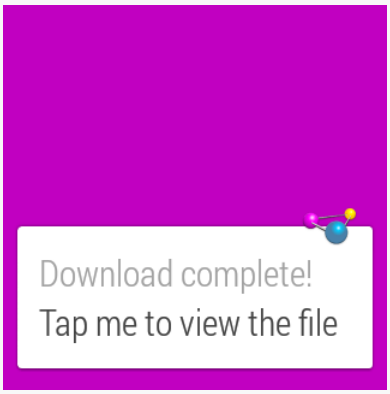


Figure 405: Pages Demo, on a Samsung Galaxy Wear, Showing Initial Notification

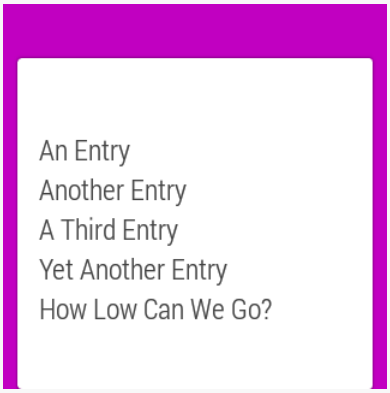


Figure 406: Pages Demo, on a Samsung Galaxy Wear, Showing Second Page

Note that you cannot use `addAction()` to define a custom action on the extra pages added to the primary `Notification`. Instead, use `addAction()` and `setContentAction()` on the `WearableExtender` to define actions associated with those extra pages. We will see this in use in the next section.

Wear-Only Actions

Sometimes, you may want certain actions to only be available on the Wear device, and not on the primary device. We will see a specific example of this coming up in the next section, when we cover voice input actions.

Sometimes, you may want a different mix of actions on the primary device versus the Wear device — some in common, some only on the primary device, some only on the Wear device.

To set up Wear-only actions, use `addAction()` on `WearableExtender`, as opposed to (or in addition to) `addAction()` on `NotificationCompat.Builder`. This takes an action as a parameter, which you create using `NotificationCompat.Action.Builder`, a custom builder for building `Notification` actions.

This is illustrated in the `Notifications/WearActions` sample project, yet another variation on the "launch an activity, show a `Notification`" samples that we have been using. This time, though, we will apply an action to the Wear device:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  NotificationManager mgr=
    (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
  }

  NotificationCompat.Builder normal=buildNormal();
  NotificationCompat.Action.Builder wearActionBuilder=
      new NotificationCompat.Action.Builder(android.R.drawable.ic_media_pause,
                                            getString(R.string.pause),
                                            buildPendingIntent(Settings.ACTION_DATE_SETTINGS));

  NotificationCompat.Builder extended=
      new NotificationCompat.WearableExtender()
          .addAction(wearActionBuilder.build())
          .extend(normal);

  NotificationManagerCompat.from(this).notify(NOTIFY_ID, extended.build());

  finish();
}
```

(from Notifications/WearActions/app/src/main/java/com/commonsware/android/wearactions/MainActivity.java)

Here, we:

- Create a `NotificationManagerCompat` instance
- Create the primary ("normal") `Notification`, using the same process as before
- Create an instance of `NotificationCompat.Action.Builder`, providing it the icon, label, and `PendingIntent` to be invoked for this action

- Create an instance of `NotificationCompat.WearableExtender`, adding the newly-defined action to it, and using the `WearableExtender` to `extend()` the primary `Notification`
- Show that extended `Notification` using the `NotificationManagerCompat` instance

However, note that we have also defined an action on the primary `Notification`:

```
private NotificationCompat.Builder buildNormal() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setAutoCancel(true)
   .setDefaults(Notification.DEFAULT_ALL)
   .setContentTitle(getString(R.string.download_complete))
   .setContentText(getString(R.string.fun))
   .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
   .setSmallIcon(android.R.drawable.stat_sys_download_done)
   .addAction(android.R.drawable.ic_media_play,
      getString(R.string.play),
      buildPendingIntent(Settings.ACTION_SETTINGS));

  return(b);
}
```

(from Notifications/WearActions/app/src/main/java/com/commonsware/android/wearactions/MainActivity.java)

`addAction()` on `WearableExtender` *replaces*, for the Wear device, any actions defined on the `Notification` itself using `addAction()`, but not the action defined via `setContentIntent()`.

On the primary device, we do not see the wear-only action:

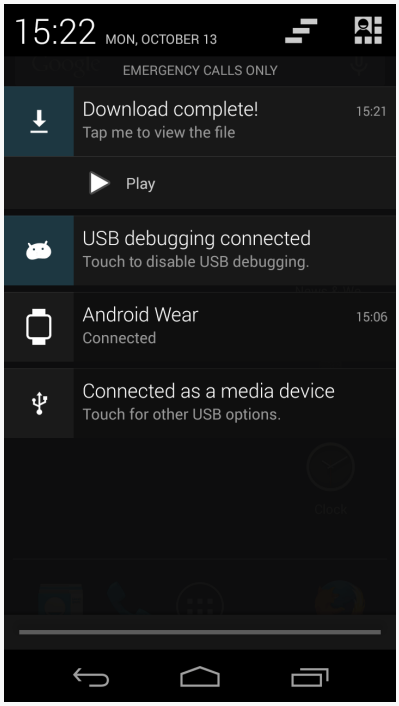


Figure 407: WearActions Demo, on a Galaxy Nexus

On a Wear device, though, we see both the wear-only and the main content action, but not the device-only action added via `addAction()` on the `NotificationCompat.Builder`:

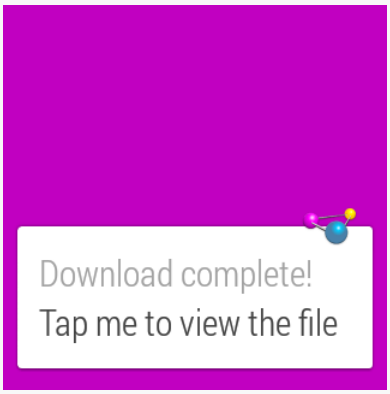


Figure 408: WearActions Demo, on a Samsung Galaxy Wear, Showing Notification

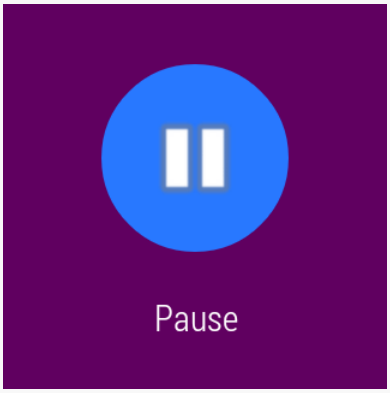


Figure 409: WearActions Demo, on a Samsung Galaxy Wear, Showing Wear-Only Action

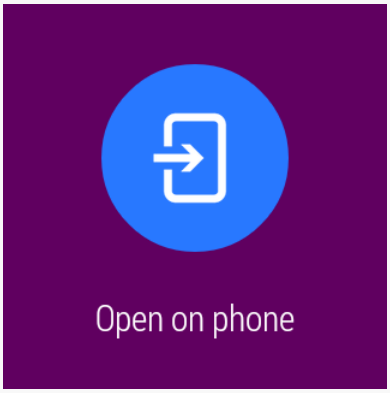


Figure 410: WearActions Demo, on a Samsung Galaxy Wear, Showing Main Content Action

Hence:

- If you want actions only on the primary device, define those before applying a `WearableExtender` and its `addAction()`
- If you want actions only on the Wear device, define those using a `WearableExtender` and its `addAction()`
- If you want the same actions on both devices, define those using both flavors of `addAction()` (on `NotificationCompat.Builder` for the primary device and on `WearableExtender` for the Wear device)

Voice Input

In the spirit of Dick Tracy's two-way wrist radio, Android Wear allows you to talk to your wrist and not seem like you are completely insane.

In particular, your `Notification`, when presented on the Wear, can request that the user provide you with a response, via voice input or via canned responses. This can be very handy:

- Responding to a text message without pulling out one's phone
- Directing your app to file an incoming email message into a particular folder or label
- Responding to a police alert, requesting your assistance, by indicating that you will be on your way as soon as you can find your bright yellow trenchcoat
- And so on

In many cases, with a regular `Notification`, the result of the user choosing an action is for us to display an activity. Sometimes, though, that's not what we want, such as a music player's `Notification` handling "pause" and similar events via its background service. Similarly, actions from a `Notification` seen on a Wear device will sometimes need to perform operations in the background, as the user may not be in position to look at your UI. This is especially true with voice input — usually, if we are bothering to dictate words to our wrist, that should happen instead of opening up the primary device. As a result, our flow for responding to the action is a little bit different, as is illustrated in the `Notifications/VoiceInput` sample project.

The Activity and Notification

Let's walk through the `MainActivity` that sets up our `Notification`:

```
package com.commonsware.android.wearvoice;

import android.app.Activity;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
```

```
import android.support.v4.app.RemoteInput;

public class MainActivity extends Activity {
  private static final String CHANNEL_WHATEVER="channel_whatever";
  private static final int NOTIFY_ID=1337;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    Intent i=new Intent(this, VoiceReceiver.class);
    PendingIntent pi=
        PendingIntent.getBroadcast(this, 0, i,
            PendingIntent.FLAG_UPDATE_CURRENT);

    RemoteInput remoteInput=
        new RemoteInput.Builder(VoiceReceiver.EXTRA_SPEECH)
          .setLabel(getString(R.string.talk))
          .setChoices(getResources().getStringArray(R.array.replies))
          .build();

    NotificationCompat.Action wearAction=
        new NotificationCompat.Action.Builder(
              android.R.drawable.ic_btn_speak_now,
              getString(R.string.talk),
              pi).addRemoteInput(remoteInput).build();

    NotificationCompat.WearableExtender wearExtender=
        new NotificationCompat.WearableExtender()
            .addAction(wearAction);

    NotificationCompat.Builder builder=
        new NotificationCompat.Builder(this, CHANNEL_WHATEVER)
            .setSmallIcon(android.R.drawable.stat_sys_download_done)
            .setContentTitle(getString(R.string.title))
            .setContentText(getString(R.string.talk))
            .extend(wearExtender);

    NotificationManagerCompat
```

```
      .from(this)
      .notify(NOTIFY_ID, builder.build());

    finish();
  }
}
```

(from Notifications/VoiceInput/app/src/main/java/com/commonsware/android/wearvoice/MainActivity.java)

We start by creating a broadcast `PendingIntent`, pointing to a `VoiceReceiver` that will respond to the voice input. We will examine this `VoiceReceiver` later in this example.

We then set up a `RemoteInput.Builder`. This is a builder-style API for defining a `RemoteInput` configuration to attach to a Wear-only action. Here, we configure it with:

- the key for retrieving the response in our `VoiceReceiver` (`VoiceReceiver.EXTRA_SPEECH`)
- the label to prompt the user for what we are looking for them to provide (an `R.string.talk` string resource)
- a `String` array of canned responses that the user can choose from rather than dictate their own answer and go through speech-to-text conversion (pulled from an `R.array.replies` `<string-array>` resource)

That `RemoteInput` is then applied to a `NotificationCompat.Action`, via its `NotificationCompat.Action.Builder` and the `addRemoteInput()` method. That `Action`, in turn, is wrapped in a `NotificationCompat.WearableExtender`, which is used to `extend()` a `NotificationCompat.Builder`.

Finally, the resulting `Notification` is raised using a `NotificationManagerCompat` instance.

The Receiver

Our `VoiceReceiver`, registered in the manifest, is set up to respond to the voice action:

```
package com.commonsware.android.wearvoice;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
```

```
import android.os.Bundle;
import android.support.v4.app.RemoteInput;
import android.util.Log;
import android.widget.TextView;

public class VoiceReceiver extends BroadcastReceiver {
  static final String EXTRA_SPEECH="speech";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    Bundle input=RemoteInput.getResultsFromIntent(i);

    if (input!=null) {
      CharSequence speech=input.getCharSequence(EXTRA_SPEECH);

      if (speech!=null) {
        Log.d(getClass().getSimpleName(), speech.toString());
      }
      else {
        Log.e(getClass().getSimpleName(), "No voice response speech");
      }
    }
    else {
      Log.e(getClass().getSimpleName(), "No voice response Bundle");
    }
  }
}
```

(from Notifications/VoiceInput/app/src/main/java/com/commonsware/android/wearvoice/VoiceReceiver.java)

It uses `RemoteInput.getResultsFromIntent(i)` to pick out the response we got from the user for this action. There are three major possibilities:

1. We did not get any response (should not happen)
2. We got a response, but for whatever reason, the decoded `Bundle` is missing our `VoiceReceiver.EXTRA_SPEECH` key (also should not happen)
3. The `CharSequence` from the `VoiceReceiver.EXTRA_SPEECH` key in the decoded `Bundle` is the user's response, whether from speech recognition or from choosing one of our canned responses

In this case, we just log the message to Logcat, but in principle you could do whatever you wanted. Just bear in mind that your UI may not be in the foreground, and that the device screen may be off entirely. It is also possible that your process will have been terminated between the time you raised the `Notification` and the user got around to responding to it from the Wear device. Hence, you should be

making few assumptions about the environment at the point when you get the voice response.

The Results

The Wear device starts off with a typical action:

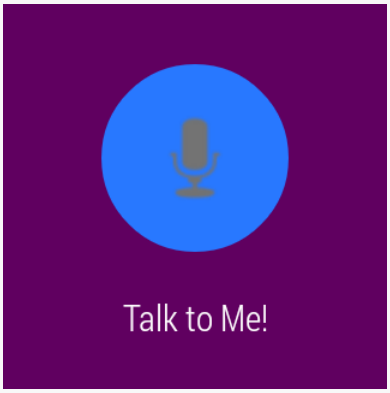


Figure 411: VoiceInput Demo, on a Samsung Galaxy Wear, Showing Voice Action

Tapping it brings up a voice input screen, where the user can dictate some text:

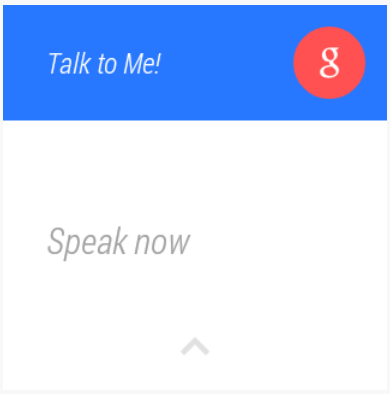


Figure 412: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input

If the user delays too long without saying anything recognizable, or if the user swipes up the screen, they are taken to our list of canned responses:

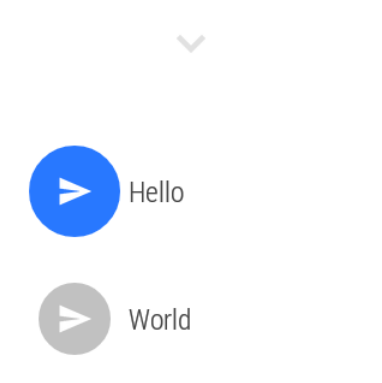


Figure 413: WearActions Demo, on a Samsung Galaxy Wear, Showing Canned Responses

If the user instead does dictate some text, initially they are shown just the interpreted text:

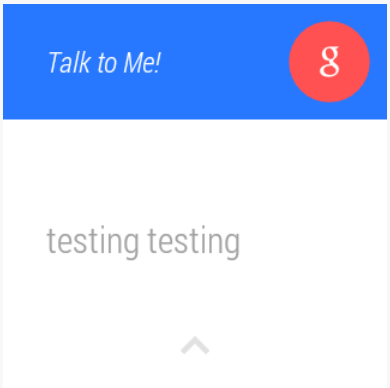


Figure 414: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input Results

Then a cancel button with a progress indicator around the edge appears:

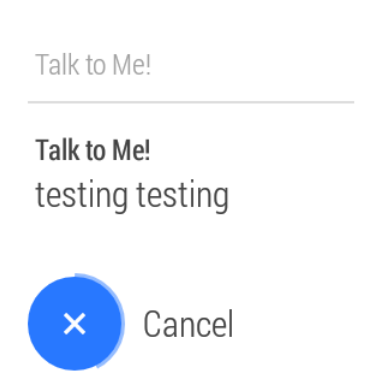


Figure 415: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input Progress

If the user taps the cancel button before the progress indicator elapses, they are prompted to confirm or reject the input:

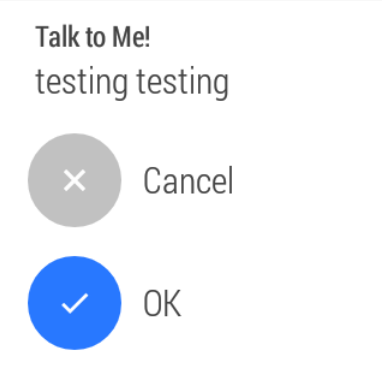


Figure 416: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input Confirmation

Remote Input, On-Device

As is noted above, Android Wear uses `RemoteInput` to get input from the user. However, historically, that capability was limited to notifications appearing on Wear.

Starting with Android 7.0, `RemoteInput` is also available for standard device notifications. Rather than using voice input, you get a small `EditText` into which the user can type something and submit it. You get what the user typed in, and can use that as needed.

The Notifications/RemoteInput sample project is a near-clone of the Notifications/VoiceInput sample project profiled in the advanced Notifications chapter. Instead of putting the RemoteInput in an action on the WearExtender, it puts the RemoteInput on the main Notification itself:

```
package com.commonsware.android.remoteinput;

import android.app.Activity;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.RemoteInput;

public class MainActivity extends Activity {
  static final String CHANNEL_WHATEVER="channel_whatever";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    Intent i=new Intent(this, RemoteInputReceiver.class);
    PendingIntent pi=
        PendingIntent.getBroadcast(this, 0, i,
          PendingIntent.FLAG_UPDATE_CURRENT);

    RemoteInput remoteInput=
        new RemoteInput.Builder(RemoteInputReceiver.EXTRA_INPUT)
          .setLabel(getString(R.string.talk))
          .build();

    NotificationCompat.Action remoteAction=
        new NotificationCompat.Action.Builder(
              android.R.drawable.ic_btn_speak_now,
```

```
              getString(R.string.talk),
              pi).addRemoteInput(remoteInput).build();

    NotificationCompat.Builder builder=
      RemoteInputReceiver.buildNotificationBase(this)
        .addAction(remoteAction);

    NotificationManagerCompat
      .from(this)
      .notify(RemoteInputReceiver.NOTIFY_ID, builder.build());

    finish();
  }
}
```

(from Notifications/RemoteInput/app/src/main/java/com/commonsware/android/remoteinput/MainActivity.java)

The activity, when launched, will raise the `Notification` with a "Talk to Me" action:

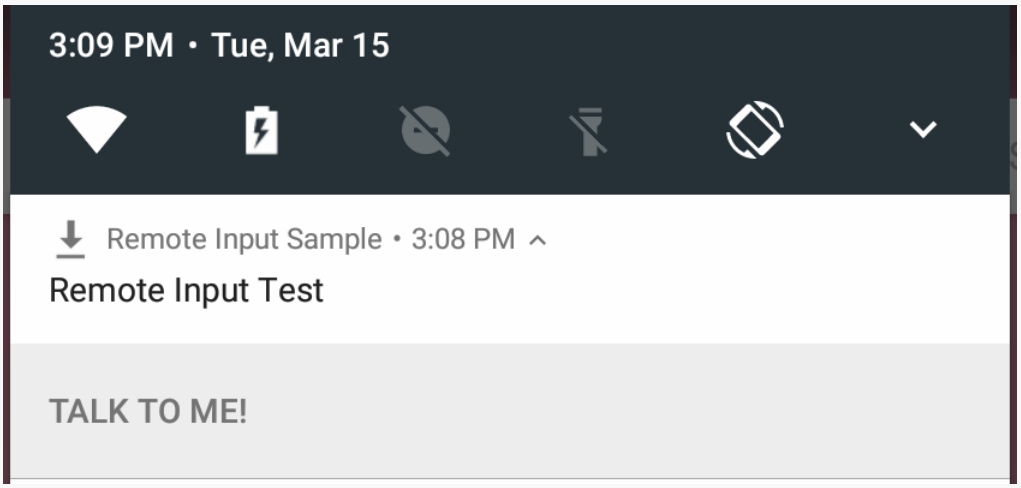


Figure 417: Notification with Action

Tapping on the action converts it into an `EditText`, with your action prompt as the hint, plus an arrow-shaped "send" button:

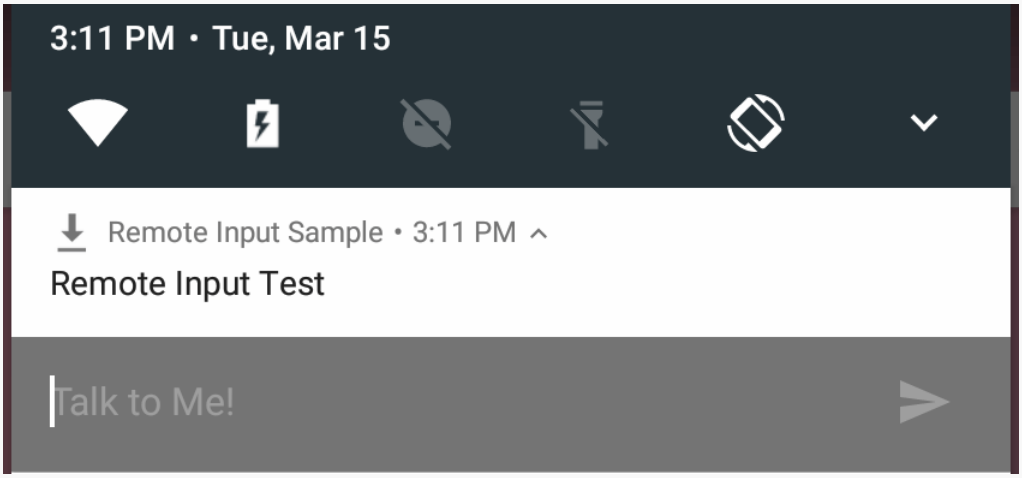


Figure 418: Notification with Remote Input

Typing something in and tapping that button converts the button into a progress spinner:

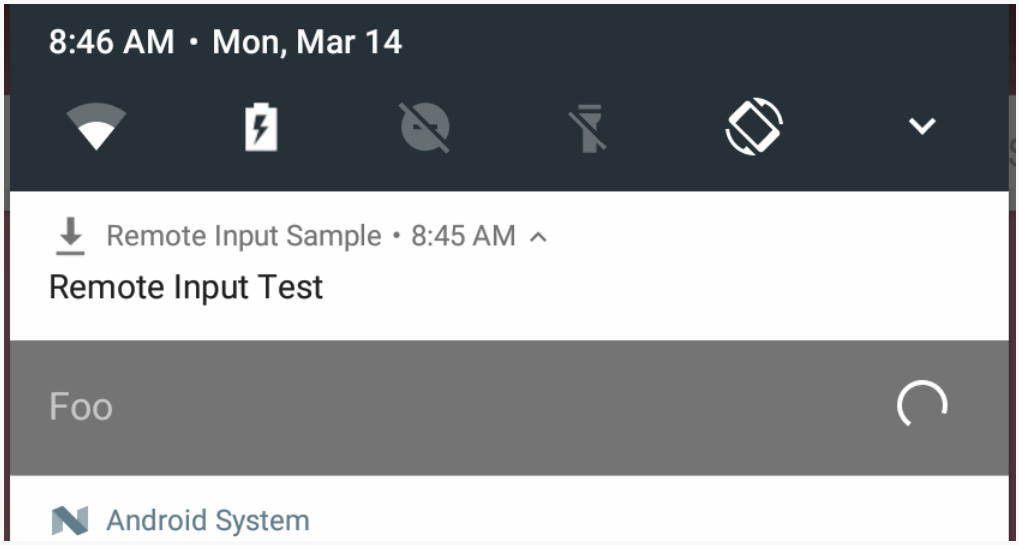


Figure 419: Notification with Remote Input and Progress Spinner

Also, the `PendingIntent` that you associated with the action is invoked. In this case, that triggers a broadcast to `RemoteInputReceiver`:

```
package com.commonsware.android.remoteinput;

import android.content.BroadcastReceiver;
import android.content.Context;
```

```
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.RemoteInput;
import android.util.Log;

public class RemoteInputReceiver extends BroadcastReceiver {
  static final int NOTIFY_ID=1337;
  static final String EXTRA_INPUT="input";

  static NotificationCompat.Builder buildNotificationBase(Context ctxt) {
    NotificationCompat.Builder builder=
      new NotificationCompat.Builder(ctxt, MainActivity.CHANNEL_WHATEVER)
        .setSmallIcon(
          android.R.drawable.stat_sys_download_done)
        .setContentTitle(ctxt.getString(R.string.title));

    return(builder);
  }

  @Override
  public void onReceive(Context ctxt, Intent i) {
    Bundle input=RemoteInput.getResultsFromIntent(i);

    if (input!=null) {
      CharSequence speech=input.getCharSequence(EXTRA_INPUT);

      if (speech!=null) {
        Log.d(getClass().getSimpleName(), speech.toString());
      }
      else {
        Log.e(getClass().getSimpleName(), "No voice response speech");
      }
    }
    else {
      Log.e(getClass().getSimpleName(), "No voice response Bundle");
    }

    NotificationCompat.Builder builder=
      buildNotificationBase(ctxt);

    NotificationManagerCompat
      .from(ctxt)
      .notify(RemoteInputReceiver.NOTIFY_ID, builder.build());
  }
}
```

(from Notifications/RemoteInput/app/src/main/java/com/commonsware/android/remoteinput/RemoteInputReceiver.java)

Here, we get what the user typed in via our designated extra (`EXTRA_INPUT`, as requested via the `RemoteInput.Builder`), which we can use as we see fit, such as logging it to Logcat.

However, we *also* have to update or cancel the `Notification`. Otherwise, that progress spinner will spin indefinitely. If the `Notification` still has value to the user after the `RemoteInput`, just update it, with or without another `RemoteInput` (depending on whether one would now be needed). You might also show the user's input in the updated `Notification`. Or, if the `Notification` is no longer needed, just `cancel()` it. In this case, we raise a fresh `Notification` for this ID, just without the `RemoteInput` that the activity added.

Note that `setChoices()` on `RemoteInput` is ignored for regular device notifications. You can use this for Wear notifications to give the user a list of strings to choose from, as an alternative to voice recognition.

You can call `setRemoteInputHistory()` on your `Builder` as well. This takes a `CharSequence` array, though many developers will elect to use a simple `String` array. This represents the inputs supplied by the user and accepted by your app, in reverse chronological order (first element in the array is the most recent input). Some of this history may be added automatically to the `Notification` when you raise the updated `Notification` containing this input history.

Notification Groups

Another concept introduced with Wear in mind was the notification group. With this, you create a summary `Notification`, along with detail `Notifications` for individual events. The quintessential example is an email app, with a summary `Notification` indicating the unread message count, and with detail `Notifications` for individual messages.

The idea for Wear was to allow the user to individually respond to the detail `Notifications` without having to pull out the associated phone or tablet. That phone or tablet would show the summary `Notification`, since the user could just tap on it and bring up the activity to see the detail.

For some reason, Google back-pedaled on that last part, as with Android 7.0, phones and tablets will also show the summary-and-detail `Notification` hierarchy.

The `Notifications/Stacked` sample project, presented in the chapter on advanced `Notifications`, demonstrates this without any modifications. Initially, the user just sees the summary:

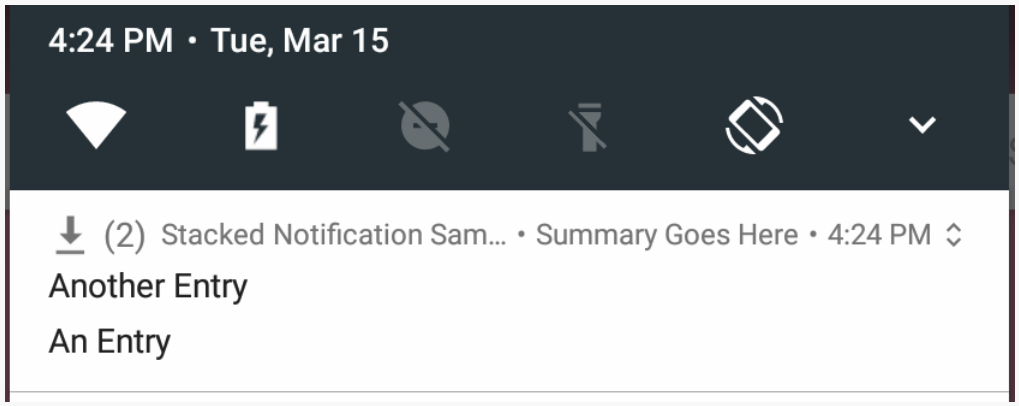


Figure 420: Stacked Notification, Showing Summary

A two-finger swipe gesture will expose the full hierarchy:

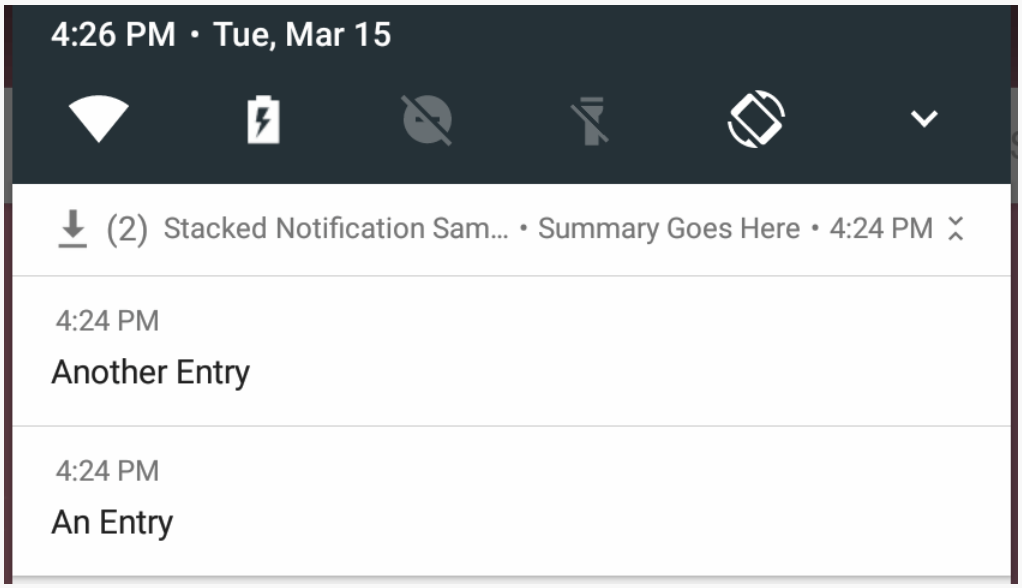


Figure 421: Stacked Notification, Showing Hierarchy

Lockscreen Notifications

Historically, notification icons would be visible on the user's lockscreen, but that was it. This would give the user an indication of what apps need attention, but no additional context.

Android 5.0 added notifications to the lockscreen, to help provide that missing context. Now users can have more details about the notifications, to determine whether it is necessary to unlock the device right now to deal with them.

Also note that on Android 7.0+, `RemoteInput` works on the lockscreen. You can see this in action with the sample app — just leave the `Notification` up, then lock the device. When you power on the screen again, you will get the lockscreen, and the `Notification` tile will appear. It will not visibly show any sign of remote input, but if you swipe down on the `Notification`, the remote input field will appear, and you can type in a message.

However, this also raises privacy concerns, as now notification text can be seen by anyone with access to the phone. As such, Android 5.0 introduced the concept of *visibility* to notifications, so developers can help control what is shown on the lockscreen versus what is shown only past the lockscreen.

However, these visibility options are only useful if:

- The device has a pattern, PIN, or password set, so it is not merely some swipe-to-unlock approach
- The user has indicated that the system should "hide sensitive notification content", either when they set up the pattern/PIN/password:

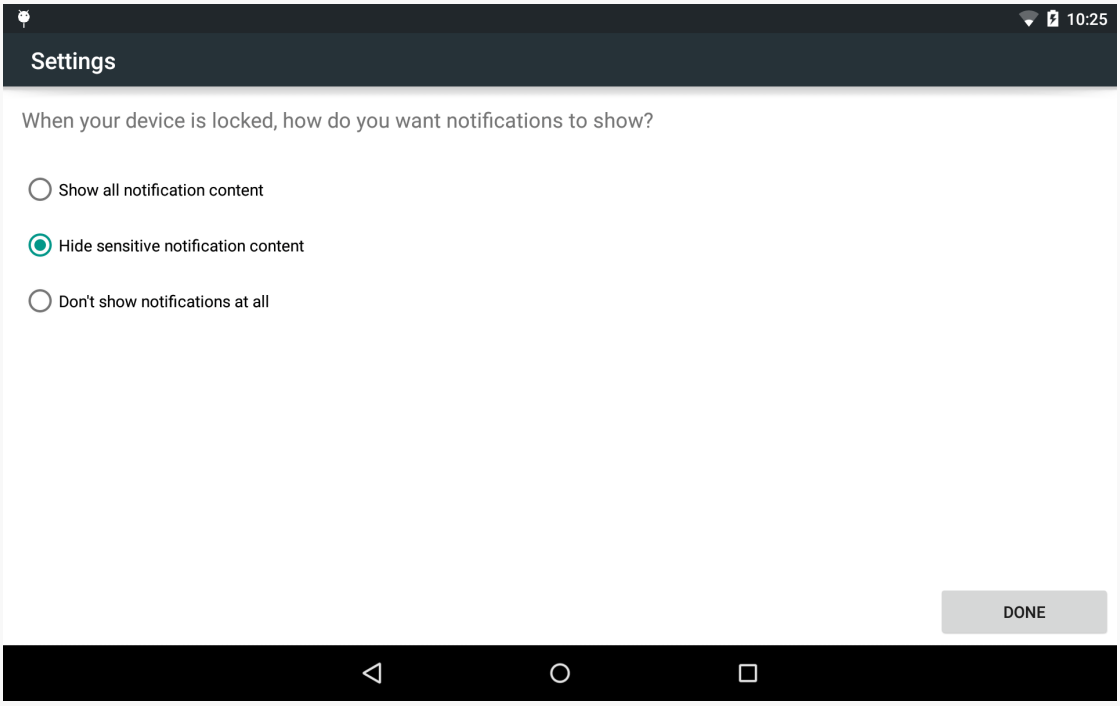


Figure 422: Choosing Notification Control, When Securing the Lockscreen

or in the “Sound & notification” portion of the Settings app:

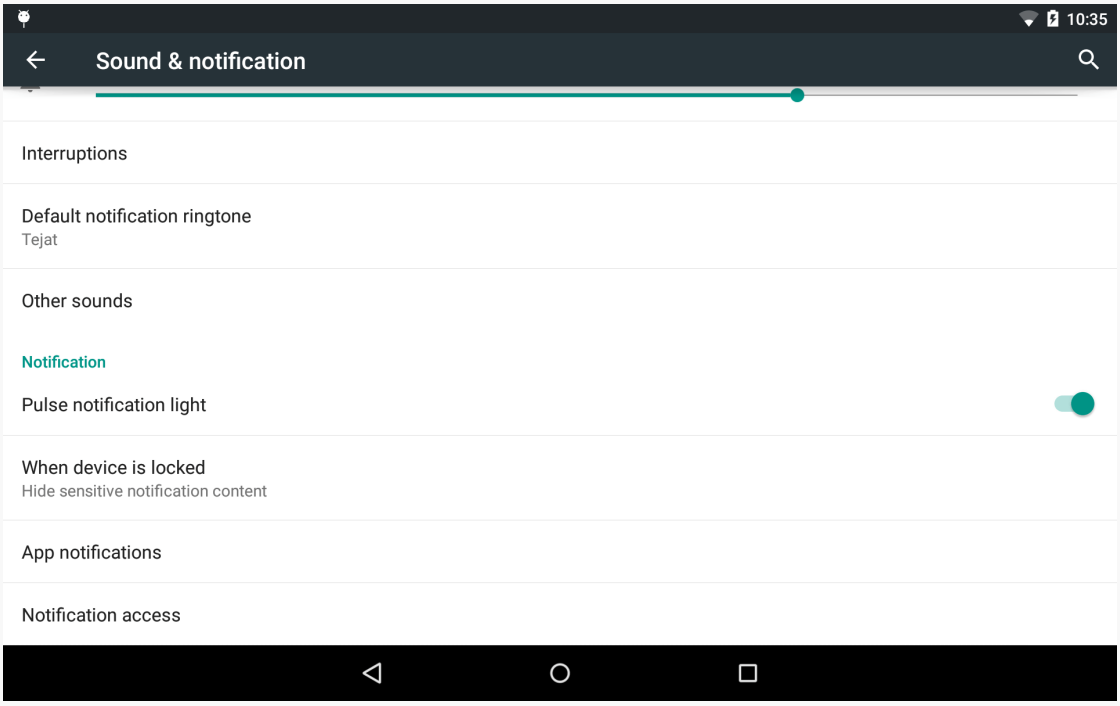


Figure 423: "Sound & notification" Settings

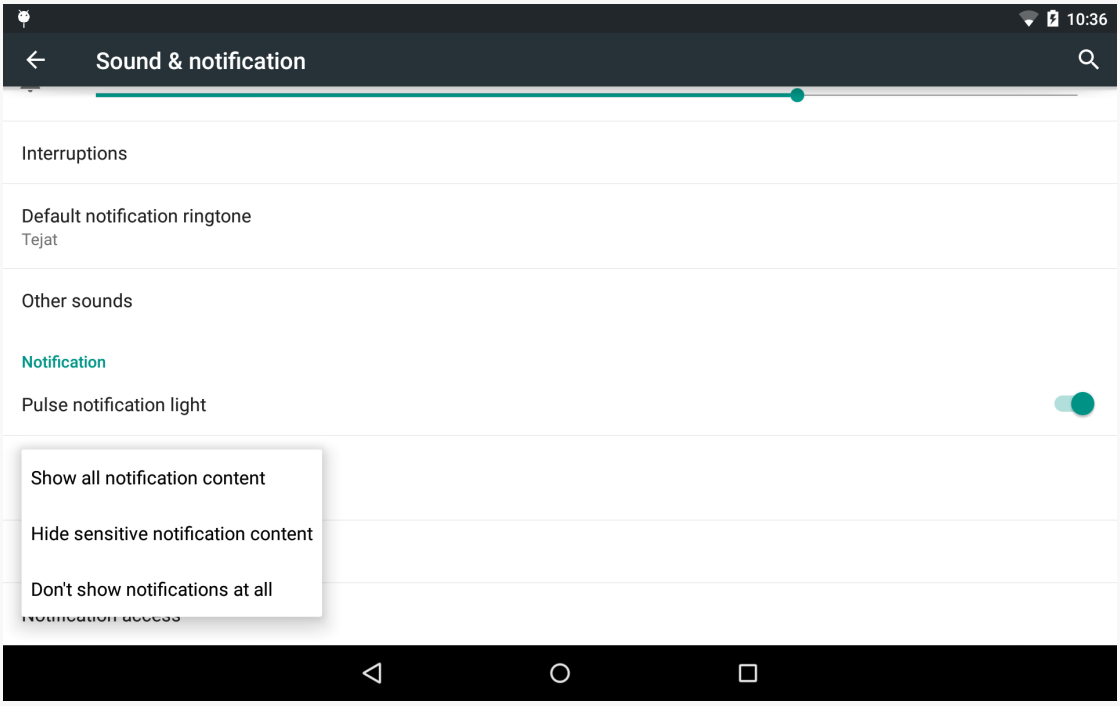


Figure 424: Notification Control Options in "Sound & notification" in Settings

Given that the user has enabled "hide sensitive notification content" mode, you as a developer can choose a visibility to apply to your notifications. There are three such visibility options — private, public, and secret — covered in the following sections.

Private Notifications

The default behavior is a "private" `Notification`. Basic information appears on the lockscreen, but not the whole `Notification`. However, you as a developer can also provide a *separate* `Notification` that *will* be shown on the lockscreen, so you can choose what information appears publicly and what information does not.

The sample app for this section has a "public" edition of the `Notification` that shows up on the lockscreen:

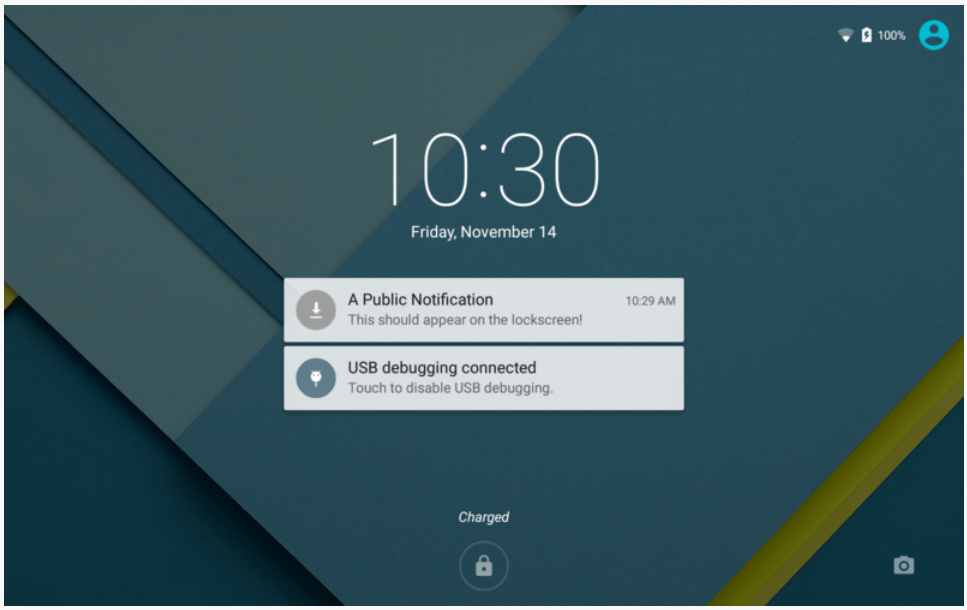


Figure 425: Public Edition of Private Lockscreen Notification, on a Nexus 7

Public Notifications

Instead of creating a separate `Notification` for public visibility on the lockscreen, you could flag your main `Notification` as having public visibility. This is suitable for notifications where there is little to no privacy implications for having the information appear on the lockscreen.

Secret Notifications

A `Notification` with visibility set to "secret" will not show up on the lockscreen at all. The ringtone, etc. will occur, as requested (and based on device settings, like it being muted), but otherwise there is no visible indication on the lockscreen that your `Notification` exists. Only when the user gets past the lockscreen will your `Notification` appear, in the status bar.

A Visibility Sample

The `Notifications/Lollipop` sample project demonstrates the use of these visibility values. It also demonstrates heads-up notifications, covered later in this chapter.

The user interface consists of a `Spinner` of possible `Notification` variants, a `SeekBar` to allow the user to specify a delay period in seconds before showing the `Notification`, and a `Button` to trigger showing the `Notification`:

```
<?xml version="1.0" encoding="utf-8"?>

<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp"
    android:stretchColumns="1">

    <TableRow>
        <TextView
            android:text="@string/type_label"/>

        <Spinner
            android:id="@+id/type"/>
    </TableRow>

    <TableRow>
        <TextView
            android:text="@string/delay_label"/>

        <SeekBar
            android:id="@+id/delay"
            android:progress="5"
            android:max="30"/>
    </TableRow>

    <Button
```

```
        android:text="@string/notify_button"
        android:id="@+id/download"
        android:onClick="notifyMe"/>
</TableLayout>
```

(from Notifications/Lollipop/app/src/main/res/layout/main.xml)

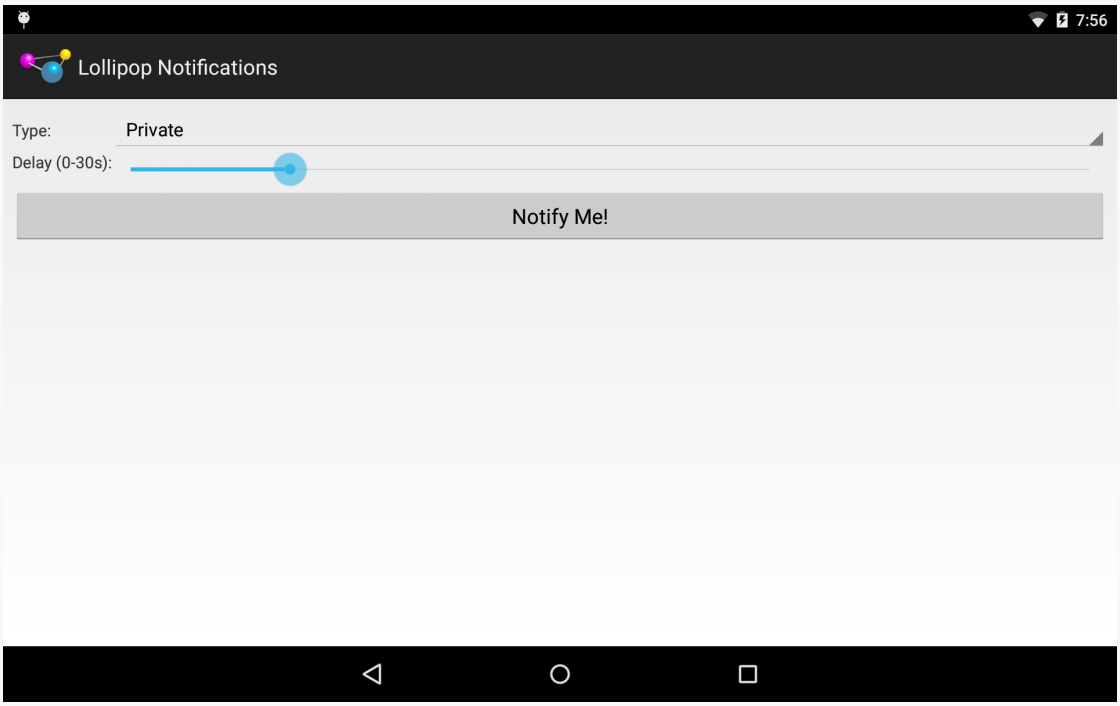


Figure 426: Lollipop Notifications Demo, on a Nexus 7

The `onCreate()` method of our launcher activity (`MainActivity`) initializes the UI:

```
package com.commonsware.android.lollipopnotify;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.SeekBar;
import android.widget.Spinner;

public class MainActivity extends Activity {
  private Spinner type=null;
  private SeekBar delay=null;

  @Override
```

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    type=(Spinner)findViewById(R.id.type);

    ArrayAdapter<String> types=
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item,
            getResources().getStringArray(R.array.types));

    types.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    type.setAdapter(types);

    delay=(SeekBar)findViewById(R.id.delay);
  }

  public void notifyMe(View v) {
    Intent i=new Intent(this, AlarmReceiver.class)
        .putExtra(AlarmReceiver.EXTRA_TYPE, type.getSelectedItemPosition());
    PendingIntent pi=PendingIntent.getBroadcast(this, 0, i,
                                            PendingIntent.FLAG_UPDATE_CURRENT);
    AlarmManager mgr=(AlarmManager)getSystemService(ALARM_SERVICE);

    mgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()+(1000*delay.getProgress()),
        pi);
  }
}
```

(from Notifications/Lollipop/app/src/main/java/com/commonsware/android/lollipopnotify/MainActivity.java)

In particular, `onCreate()` populates the `Spinner` based on a `<string-array>` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="types">
        <item>Private</item>
        <item>Public</item>
        <item>Secret</item>
        <item>Heads-Up</item>
    </string-array>
</resources>
```

(from Notifications/Lollipop/app/src/main/res/values/arrays.xml)

When the button is clicked, the `notifyMe()` method on `MainActivity` is called. Here, we:

- Create an `Intent` pointing at an `AlarmReceiver`
- Package an extra on the `Intent` that contains the selected position of the `Spinner`

- Wrap the `Intent` in a `getBroadcast()` `PendingIntent`
- Use `set()` on `AlarmManager` to invoke the `PendingIntent` after the delay period specified via the `SeekBar`

Since the `targetSdkVersion` of this project is below 19, the `set()` method will behave in an exact fashion, triggering our `AlarmReceiver` at the designated time.

`AlarmReceiver`, in turn, uses a `switch` statement to call out to different private methods based upon which `Spinner` item was selected:

```
package com.commonsware.android.lollipopnotify;

import android.app.Notification;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.provider.Settings;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;

public class AlarmReceiver extends BroadcastReceiver {
  private static final String CHANNEL_WHATEVER="channel_whatever";
  private static final String CHANNEL_HEADS_UP="channel_heads_up";
  private static final int NOTIFY_ID=1337;
  static final String EXTRA_TYPE="type";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    NotificationManager mgr=
      (NotificationManager)ctxt.getSystemService(Context.NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_HEADS_UP,
        "Heads Up!", NotificationManager.IMPORTANCE_HIGH));
    }

    NotificationManagerCompat mgrCompat=NotificationManagerCompat.from(ctxt);

    switch (i.getIntExtra(EXTRA_TYPE, -1)) {
      case 0:
        notifyPrivate(ctxt, mgrCompat);
        break;

      case 1:
        notifyPublic(ctxt, mgrCompat);
        break;

      case 2:
```

```
      notifySecret(ctxt, mgrCompat);
      break;

    case 3:
      notifyHeadsUp(ctxt, mgrCompat);
      break;
  }
}

private void notifyPrivate(Context ctxt, NotificationManagerCompat mgr) {
  Notification pub=
    buildBase(ctxt, CHANNEL_WHATEVER, R.string.public_title).build();

  mgr.notify(NOTIFY_ID,
    buildBase(ctxt, CHANNEL_WHATEVER, R.string.private_title).setPublicVersion(pub).build());
}

private void notifyPublic(Context ctxt, NotificationManagerCompat mgr) {
  mgr.notify(NOTIFY_ID,
      buildBase(ctxt, CHANNEL_WHATEVER, R.string.public_title)
          .setVisibility(NotificationCompat.VISIBILITY_PUBLIC)
          .build());
}

private void notifySecret(Context ctxt, NotificationManagerCompat mgr) {
  mgr.notify(NOTIFY_ID,
      buildBase(ctxt, CHANNEL_WHATEVER, R.string.secret_title)
          .setVisibility(NotificationCompat.VISIBILITY_SECRET)
          .build());
}

private void notifyHeadsUp(Context ctxt, NotificationManagerCompat mgr) {
  mgr.notify(NOTIFY_ID,
      buildBase(ctxt, CHANNEL_HEADS_UP, R.string.headsup_title)
          .setPriority(NotificationCompat.PRIORITY_HIGH)
          .build());
}

private NotificationCompat.Builder buildBase(Context ctxt, String channel,
                                             int titleId) {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(ctxt, channel);

  b.setAutoCancel(true)
      .setDefaults(Notification.DEFAULT_ALL)
      .setContentTitle(ctxt.getString(titleId))
      .setContentIntent(buildPendingIntent(ctxt, Settings.ACTION_SECURITY_SETTINGS))
      .setSmallIcon(android.R.drawable.stat_sys_download_done)
      .addAction(android.R.drawable.ic_media_play,
          ctxt.getString(R.string.play),
          buildPendingIntent(ctxt, Settings.ACTION_SETTINGS));

  return(b);
}

private PendingIntent buildPendingIntent(Context ctxt, String action) {
  Intent i=new Intent(action);

  return(PendingIntent.getActivity(ctxt, 0, i, 0));
```

```
  }
}
```

(from Notifications/Lollipop/app/src/main/java/com/commonsware/android/lollipopnotify/AlarmReceiver.java)

If the user chooses the "Private" option in the `Spinner`, we call `notifyPrivate()`. That method builds two `Notification` objects: the regular one and a separate public edition. We attach the public edition to the regular `Notification` via a call to `setPublicVersion()` on the `NotificationCompat.Builder`. Then, we raise the regular `Notification`. This will show the public edition if the lockscreen is locked; otherwise, it will show the regular edition.

If the user chooses the "Public" option, we call `notifyPublic()`. That, in turn, calls `setVisibility(NotificationCompat.VISIBILITY_PUBLIC)` on the `NotificationCompat.Builder`, causing our `Notification` to appear normally both on the lockscreen and past the lockscreen.

If the user chooses the "Secret" option, we call `notifySecret()`. That uses `setVisibility(NotificationCompat.VISIBILITY_SECRET)` to configure the `Notification` to only appear once the user has gotten past the lockscreen.

The "Heads-Up" option — fourth in the `Spinner` — is covered in the next section.

Priority, Importance, and Heads-Up Notifications

Notifications can have a priority associated with them. Normally, notifications with higher priority will appear higher in the list of notifications in the notification tray than will notifications with lower priority.

Android 5.0 took this a step further, showing high-priority notifications in a "heads-up" style, popping up a small dialog-like window over the main screen, with the same basic content as would appear for the `Notification` in its tile in the notification tray:

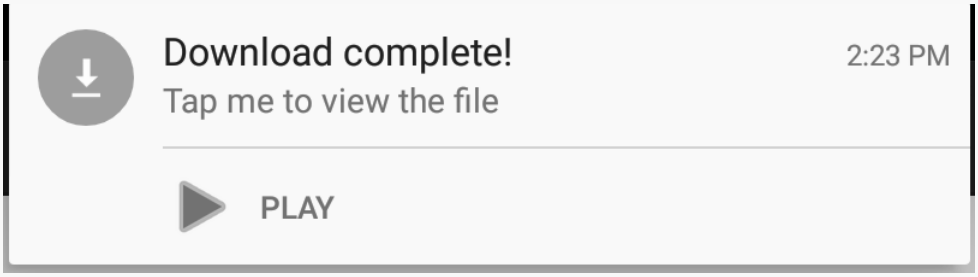


Figure 427: Lollipop Demo, on a Nexus 7, Showing Heads-Up Notification

Users can interact with the heads-up `Notification` or ignore it; in the latter case, the `Notification` will move into the status bar and the "heads-up" display will disappear from the screen.

Note that the "priority" concept being described here seems to be independent of the notion of "priority notifications" in the user's interruption configuration in Settings. There, "priority notifications" is tied to the app, not tied to any sort of configuration of the `Notification` itself.

Android 8.0's notification channels introduced a separate concept, called importance, set on the channel. Roughly speaking, importance on a channel is the same as priority on a notification. Priorities are marked as deprecated on API Level 26+, though they still work.

Specifying the Priority

`NotificationCompat.Builder` has a `setPriority()` method that allows you to specify your requested priority. There are five priority values accepted as a parameter, all defined as constants out on the `NotificationCompat` class:

- `PRIORITY_MAX`
- `PRIORITY_HIGH`
- `PRIORITY_DEFAULT`
- `PRIORITY_LOW`
- `PRIORITY_MIN`

The actual priority applied to the `Notification` will depend upon other factors, and so you should not assume that your requested value will be accepted and applied as-is.

Results on Android 5.x Devices

The heads-up `Notification` appears as shown in the above screenshot. The pop-up itself is centered across the top of the screen, as shown below:

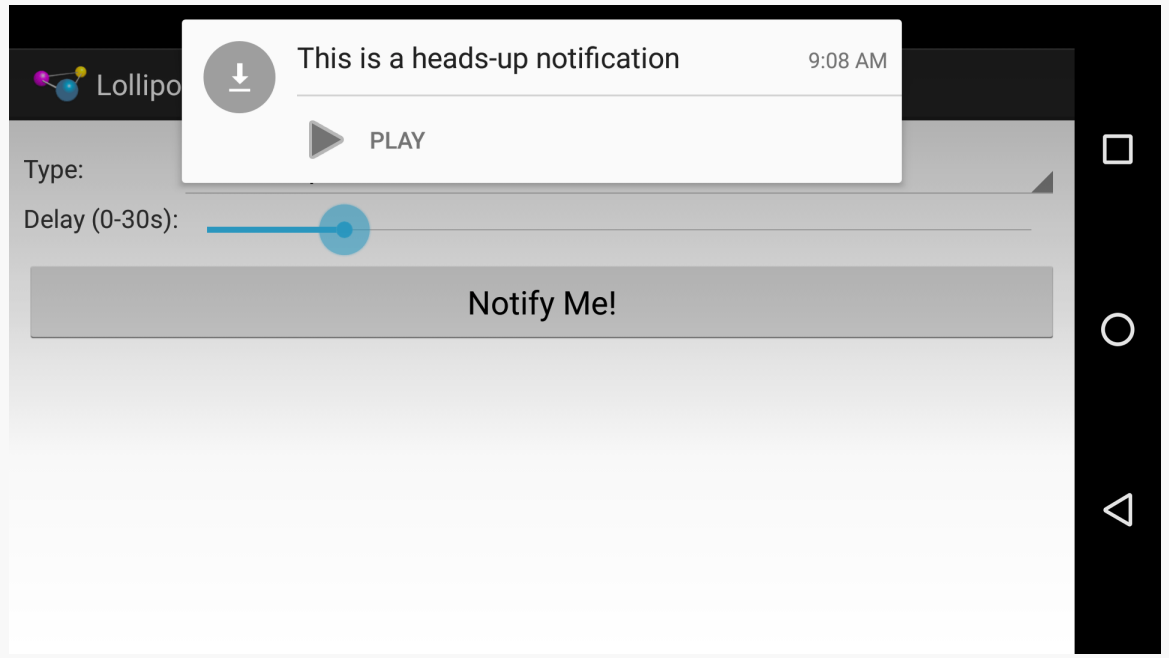


Figure 428: Lollipop Demo, Showing Heads-Up Notification

After a few seconds of inactivity, the pop-up vanishes, and the `Notification` goes into the status bar.

Results on Older Devices

The concept of priority was introduced in API Level 16 (Android 4.1). On Android 4.1 through 4.4, the only effect of priority was to help influence the sort order of notifications in the notification tray, with higher-priority items drifting towards the top.

While `NotificationCompat.Builder` will allow you to specify a priority even on devices running older versions of Android than 4.1, the requested priority will be ignored, simply because priority did not exist back then. Hence, while your code will still work, it will have no effect on such old devices.

Full-Screen Notifications

Before Android 5.0 added heads-up notifications, while priority would influence things like sort order, it would have no real impact on how the user would be informed about whatever event triggered the `Notification`. The user would still just get an icon in the status bar, and perhaps a ringtone and other hardware output.

However, sometimes we need to be somewhat more "in the user's face", such as for a calendar event reminder, or for an incoming phone call from our VOIP app.

It is tempting to launch an activity in these cases. In fact, that is what the user tends to perceive as happening, on Android 4.4 and older devices. And some apps no doubt actually do launch an activity.

A "middle ground" between showing a `Notification` and launching an activity is to use a full-screen `Notification`. Here, we provide a `PendingIntent` that should be invoked if the user is actively using the device at the time of the `Notification`. Typically, that `PendingIntent` will display an activity. However, on Android 5.0+, the behavior has changed, where a full-screen `Notification` actually just triggers a heads-up notification, as would a high-priority `Notification`.

Requesting Full-Screen Output

Prior to Android 8.0, all that you need to do to set up a `Notification` to be full-screen is to call `setFullScreenIntent()` on your `NotificationCompat.Builder`, supplying two values:

1. A `PendingIntent` to be invoked when the notification is added to the screen
2. A `boolean`, where `true` indicates that even if the user has blocked notifications, you want this one to appear

For example, in the [Notifications/FullScreen]() sample project, `MainActivity` shows a `Notification` constructed via the `buildNormal()` method:

```
private NotificationCompat.Builder buildNormal() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setAutoCancel(true)
   .setDefaults(Notification.DEFAULT_ALL)
   .setContentTitle(getString(R.string.download_complete))
   .setContentText(getString(R.string.fun))
   .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
```

```
    .setSmallIcon(android.R.drawable.stat_sys_download_done)
     .setNumber(5)
    .setFullScreenIntent(buildPendingIntent(Settings.ACTION_DATE_SETTINGS), true)
    .addAction(android.R.drawable.ic_media_play,
        getString(R.string.play),
        buildPendingIntent(Settings.ACTION_SETTINGS));

  return(b);
```

(from Notifications/FullScreen/app/src/main/java/com/commonsware/android/fullscreen/MainActivity.java)

Here, the `PendingIntent` is created using the same `buildPendingIntent()` method as before, this time opening up a distinct screen from the Settings app.

Results on Android 5.0-7.1 Devices

On Android 5.0 through 7.1, the "full screen" `Notification` appears as a heads-up `Notification`:

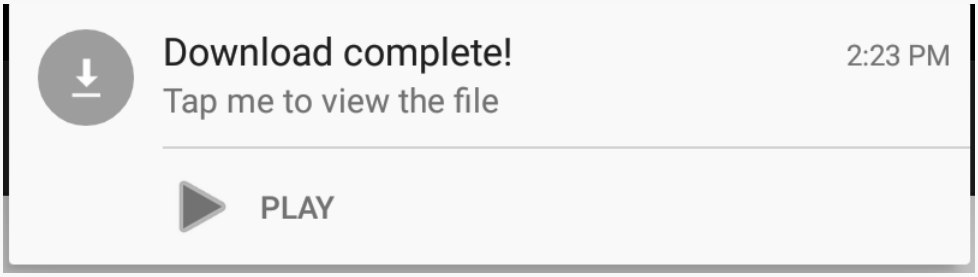


Figure 429: FullScreen Demo, on a Nexus 7, Showing "Full Screen" Notification

Note that there is no obvious way to actually invoke the `PendingIntent` associated with the `setFullScreenIntent()` method. Hence, you need to make sure that the `Notification` has some other means of getting the user to the right place in your UI, such as via `setContentIntent()` or an action.

Results on Android 3.0-4.4 Devices

On API Levels 11 through 19 (Android 3.0 through 4.4), the effect of a full-screen `PendingIntent` is to invoke the `PendingIntent` when the `Notification` is added to the screen. This will happen regardless of whether the user is using the device or not, though if the device is asleep, the activity triggered by the `PendingIntent` will only be visible once the user gets past their lockscreen.

Note that the `Notification` is *also* shown, along with whatever the `PendingIntent` does. That `Notification` is not automatically cleared when the user exits out of that

activity via BACK, HOME, etc. Hence, it is up to you to clear that `Notification` if and when it is no longer relevant. The primary value of the `Notification` is to have the icon appear in the status bar on the lockscreen — even though the user cannot interact with your `Notification` then, the user may recognize your icon and therefore elect to unlock their device to see what all the fuss is about.

Results on Older Devices

Full-screen notifications were not supported prior to Android 3.0. While `NotificationCompat.Builder` will allow you to call `setFullScreenIntent()`, the value will be ignored prior to API Level 11.

In theory, there is nothing stopping `NotificationCompat` from launching an activity itself, in addition to displaying the `Notification`. However, at least at this time, it is not doing so, and it is fairly likely that Google will not add this in at this point.

Hence, the only way to do a "full-screen notification" is for your app to launch the desired activity, in addition to (or instead of) showing the `Notification`.

Hey, What About Android 8.0+?

On these devices, `setFullScreenIntent()` has no effect.

Progress Notifications

Often, you will see a `Notification` with a `ProgressBar` in it, showing progress of some long-running background work, such as a large download. There are two approaches towards building this sort of thing:

1. Create a custom `Notification`, as we will cover later in this chapter
2. Use `setProgress()` on the `NotificationCompat.Builder`, periodically updating the `Notification` to reflect the now-current amount of progress

Needless to say, the second option is simpler.

The `HTTP/OkHttpProgress` sample project demonstrates how this works in a fairly realistic situation: tracking progress of a long download. In this case, we will use OkHttp3, showing how you can integrate its somewhat convoluted "interceptor" API to find out about download progress, then use that information to update a `Notification`.

The UI

This sample app is a variation on other download samples shown elsewhere in the book. We have a fragment with a large "Do the Download" button. When the button is clicked, we want to start a `Downloader` service to do the actual downloading on a background thread.

The two primary differences in this fragment's `onClick()` method are:

- We are downloading a much bigger file than before — Version 1.1 of this book, from 2008. This is so we can actually see the progress move; with a short download, the download might complete before we get a chance to look at the `Notification`.
- We `finish()` the activity, to emphasize the fact that our ongoing UI is being handled by the `Notification`:

```
@Override
public void onClick(View v) {
  Intent i=new Intent(getActivity(), Downloader.class);

  i.setDataAndType(Uri.parse("http://commonsware.com/Android/Android-1_1-CC.pdf"),
                   "application/pdf");

  getActivity().startService(i);
  getActivity().finish();
}
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/DownloadFragment.java)

The Downloader Service

The more significant changes come in the `Downloader` service. Previous editions of this sample use `HttpURLConnection`, but here we switch to OkHttp3, which offers a cleaner way to find out our download progress. Plus, our foreground service `Notification` will employ the `ProgressBar` to show how far along we are in downloading the file.

Everything but the Icky Parts

A large chunk of our `Downloader IntentService` does the same stuff as you see in the `Notifications/Foreground` sample project.

Of particular note here, we call `startForeground()`, to elevate our process priority while the download is happening and show a `Notification` along the way:

```
    String filename=i.getData().getLastPathSegment();
    final NotificationCompat.Builder builder=
      buildForeground(filename);

    startForeground(FOREGROUND_ID, builder.build());
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

The `Notification` itself comes from `buildForeground()`, which takes the name of the file as a parameter and builds a `Notification` with that information:

```
private NotificationCompat.Builder buildForeground(
  String filename) {
  NotificationCompat.Builder b=new NotificationCompat.Builder(this);

  b.setContentTitle(getString(R.string.downloading))
   .setContentText(filename)
   .setSmallIcon(android.R.drawable.stat_sys_download)
   .setTicker(getString(R.string.downloading))
   .setOngoing(true);

  return(b);
}
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

Eventually, we start using `OkHttpClient` to download the file:

```
    OkHttpClient client=new OkHttpClient.Builder()
      .addNetworkInterceptor(nightTrain)
      .build();
    Request request=
      new Request.Builder().url(i.getData().toString()).build();
    Response response=client.newCall(request).execute();
    String contentType=response.header("Content-type");
    BufferedSink sink=Okio.buffer(Okio.sink(new File(output.getPath())));

    sink.writeAll(response.body().source());
    sink.close();

    stopForeground(true);
    raiseNotification(contentType, output, null);
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

We start off by building an instance of `OkHttpClient` using an `OkHttpClient.Builder`. We will take a closer look at the `addNetworkInterceptor()`

call shortly, as that is where we are hooking in our code to find out about the progress of the HTTP request.

We then:

- Create a `Request` to `GET` our file, using the path supplied to use via `getData()` on the `Intent` passed into `onHandleIntent()`
- Start executing the HTTP operation
- Capture the `Content-type` header, for use when constructing an `ACTION_VIEW` `Intent` to view the downloaded file
- Use Okio (the generic I/O subsystem underlying OkHttp3) to create a `BufferedSink` on our desired output location
- Use Okio to copy all the data from the HTTP response to that output file
- `close()` the output file
- Mark the service as no longer being foreground (via `stopForeground()`)
- Show a download-complete `Notification`

Other than using OkHttp3, little of that is different from the original foreground service sample. Where things start to get interesting is in that `addNetworkInterceptor()` call.

The Interceptor

Interceptors are a way for you to hook into the flow of OkHttp3 processing, such that your code gets invoked for any request made of this `OkHttpClient`. In particular, a network interceptor allows you to get control during the actual network I/O of processing the request.

The OkHttp Git repository contains some sample code that uses a network interceptor to track download progress, and that code forms the foundation of what is shown in this sample.

The `addNetworkInterceptor()` call in `Downloader` is using a local `Interceptor` object, named `nightTrain`:

```
    Interceptor nightTrain=new Interceptor() {
      @Override
      public Response intercept(Chain chain)
        throws IOException {
        Response original=chain.proceed(chain.request());
        Response.Builder b=original
          .newBuilder()
```

```
            .body(
              new ProgressResponseBody(original.body(),
                  progressListener));

          return(b.build());
        }
      };
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

An `Interceptor` will be called with a `Chain`, representing the HTTP request and response. The job of the `Interceptor` is three-fold:

1. Call `chain.proceed()` at some point, to kick off the actual HTTP processing
2. Return a `Response` object that will be used as the "real" response of this request
3. Do whatever work the `Interceptor` was designed to do, such as request logging

The `chain.proceed()` call returns the `Response` that would be what OkHttp3 would use in the absence of this `Interceptor`. The `Interceptor` can either return that `Response` or some other `Response`. If you want to monitor the actual network I/O — such as we want to do here, to see how many bytes we have downloaded — the recipe is to use the wrapper pattern and wrap something from the original `Response` in a wrapper that has your business logic.

That is what this sample does. `original.newBuilder()` gives us a `Response.Builder` that is based on the original `Response`. The `body()` of a `Response` is a `ResponseBody` that manages an Okio `Source` object, which handles the actual streaming. We wrap the original `ResponseBody` in a `ProgressResponseBody` that will track our download progress, put that `ProgressResponseBody` into the `Response.Builder`, then return the `Response` that is built by that `Builder`. The net effect is that all calls to the `ResponseBody` will go to our `ProgressResponseBody`.

The ProgressResponseBody

What we *really* want to wrap is the `Source`, an Okio object that is responsible for the real streaming. However, to get there, we have to wrap that original `ResponseBody` in a `ProgressResponseBody`.

ProgressResponseBody itself extends from a ResponseBodyWrapper, which is a ResponseBody that forwards everything onto a wrapped ResponseBody... except for a hook to allow us to wrap the Source:

```
package com.commonsware.android.okhttp3.progress;

import okhttp3.MediaType;
import okhttp3.ResponseBody;
import okio.BufferedSource;
import okio.Okio;
import okio.Source;

// inspired by https://github.com/square/okhttp/blob/master/samples/guide/src/main/java/okhttp3/recipes/
Progress.java

abstract class ResponseBodyWrapper extends ResponseBody {
  abstract Source wrapSource(Source original);

  private final ResponseBody wrapped;
  private BufferedSource buffer;

  ResponseBodyWrapper(ResponseBody wrapped) {
    this.wrapped=wrapped;
  }

  @Override
  public MediaType contentType() {
    return(wrapped.contentType());
  }

  @Override
  public long contentLength() {
    return(wrapped.contentLength());
  }

  @Override
  public BufferedSource source() {
    if (buffer==null) {
      buffer=Okio.buffer(wrapSource(wrapped.source()));
    }

    return(buffer);
  }
}
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/ResponseBodyWrapper.java)

Subclasses of ResponseBodyWrapper need to implement wrapSource() to wrap the Source of the Response.

ProgressResponseBody does just that, wrapping the Source in a subclass of ForwardingSource named ProgressSource:

```
package com.commonsware.android.okhttp3.progress;
```

```
import java.io.IOException;
import okhttp3.ResponseBody;
import okio.Buffer;
import okio.ForwardingSource;
import okio.Source;

// inspired by https://github.com/square/okhttp/blob/master/samples/guide/src/main/java/okhttp3/recipes/
Progress.java

class ProgressResponseBody extends ResponseBodyWrapper {
  private final Listener listener;

  ProgressResponseBody(ResponseBody wrapped, Listener listener) {
    super(wrapped);

    this.listener=listener;
  }

  @Override
  Source wrapSource(Source original) {
    return(new ProgressSource(original, listener));
  }

  class ProgressSource extends ForwardingSource {
    private final Listener listener;
    private long totalRead=0L;

    public ProgressSource(Source delegate, Listener listener) {
      super(delegate);

      this.listener=listener;
    }

    @Override
    public long read(Buffer sink, long byteCount)
      throws IOException {
      long bytesRead=super.read(sink, byteCount);
      boolean done=(bytesRead==-1);

      if (!done) {
        totalRead+=bytesRead;
      }

      listener.onProgressChange(totalRead,
        ProgressResponseBody.this.contentLength(), done);

      return(bytesRead);
    }
  }

  interface Listener {
    void onProgressChange(long bytesRead, long contentLength,
                          boolean done);
  }
}
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/ProgressResponseBody.java)

Our `ProgressSource` does two things:

1. It tracks the total number of bytes that have been read so far
2. Every time that we read more data, we call a `ProgressResponseBody.Listener` with the number of bytes that have been read so far, the known content length of the stream, and whether we are now done reading from the stream

Updating the Notification

Back in `Downloader`, the `nightTrain` passed in a `progressListener` to the `ProgressResponseBody` constructor. That `progressListener` is an implementation of `ProgressResponseBody.Listener`, where we can actually update our `Notification`:

```
    final ProgressResponseBody.Listener progressListener=
      new ProgressResponseBody.Listener() {
        long lastUpdateTime=0L;

        @Override
        public void onProgressChange(long bytesRead,
                                     long contentLength,
                                     boolean done) {
          long now=SystemClock.uptimeMillis();

          if (now-lastUpdateTime>1000) {
            builder.setProgress((int)contentLength,
              (int)bytesRead, false);
            mgr.notify(FOREGROUND_ID, builder.build());
            lastUpdateTime=now;
          }
        }
      };
```

(from HTTP/OkHttpProgress/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

Mostly, what we do is call `setProgress()` on the `NotificationCompat.Builder`. This takes:

- the maximum value of the `ProgressBar`, for which we use the length of the content
- the current progress that we have made, for which we use the number of bytes already downloaded
- whether the `ProgressBar` should be indeterminate, for which we pass `false` to get a `ProgressBar` that shows actual progress

Then, we `build()` a fresh `Notification` from the `Builder` and pass that to `notify()` with the same ID. This will update our existing `Notification`, showing the updated progress.

However, our listener is going to be invoked fairly frequently, so much that we might swamp the system just constantly updating the `Notification`. To help with that, we track when we update the `Notification` and only update it again if a second has passed.

What If We Had an Activity in the Foreground?

You may want to be presenting the progress of the download in two places: the `Notification` and the UI of your application, if a relevant bit of that UI happens to be in the foreground. For example, you might have a fragment that contained the button or action bar item that kicked off the download. So long as that fragment is visible, you might want to have a `ProgressBar` on it and update the progress there.

If you use an event bus, you can have your `ProgressResponseBody.Listener` post an event with the progress of the download. However, that event should include some sort of unique identifier for the download itself, in addition to the progress. That way, only the fragment related to this specific download will show the progress, not similar fragments elsewhere.

The APK edition of this book takes this approach in the Community Theater area, where you can download and watch presentations on Android app development topics ("appinars"). An appinar is a ZIP archive up on a CommonsWare server. When you browse the catalog of appinars and choose one, the screen will either let you download the appinar or play the appinar (if it is already downloaded). The download will use code reminiscent of what you see in the sample app, with the added feature of event bus messages publishing the progress to the rest of the app. However, we only show the download progress on the fragment for the particular appinar that is being downloaded. If, while the download is progressing, you visit other appinar descriptions, we do not want to show the download progress there, as you will think that you are downloading those appinars too. The `Notification` shows the name of the appinar that you are downloading, so it provides built-in context for what appinar the progress pertains to.

Custom Views

When you specify a title and a description for a `Notification`, you are implicitly telling Android to use a stock layout for the structure of the `Notification` object's entry in the notification drawer. However, instead, you can provide Android with the layout to use and its contents, by means of a `RemoteViews`. In other words, by using the same techniques that you use to create app widgets, you can create tailored notification drawer content. Just create the `RemoteViews` and supply it to your `NotificationCompat.Builder` via `setContent()`.

To update the notification tile content, you update your `RemoteViews` in your `Notification` and re-raise the `Notification` via a call to `notify()`. Android will apply your revised `RemoteViews` to the notification drawer content, and the user will see the changed widgets.

The `Notifications/CustomView` sample project is a clone of the `HTTP/OkHttpProgress` shown in the previous section. The difference is that we will use our own custom layout and a `RemoteViews` rather than use the standard `Notification` UI.

The Notification Layout

This sample app has its `minSdkVersion` set to 21, so we only need to worry about providing a layout that looks OK on Android 5.0+ devices. Google dramatically changed the look of notifications with Android 5.0, so a layout that looks good on older devices may not blend in well with newer devices. If you have a need to support a wider range of Android versions, you will want to consider using versioned layout resources (e.g., `res/layout/` for older devices, `res/layout-v21/` for API Level 21+ devices).

But, since this app's scope is limited, we can directly refer to `Theme.Material`-based themes, to get a layout that has elements that resembles the actual notification tile content:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
```

```
    android:id="@android:id/title"
    style="@android:style/TextAppearance.Material.Notification.Title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:ellipsize="marquee"
    android:singleLine="true" />

  <ProgressBar
    android:id="@android:id/progress"
    style="@android:style/Widget.Material.ProgressBar.Horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:indeterminate="false" />
</LinearLayout>
```

(from Notifications/CustomView/app/src/main/res/layout/notif_content.xml)

Here, we are just showing a title and a `ProgressBar`. The `TextView` uses `@android:style/TextAppearance.Material.Notification.Title`, which is the same style as is used by the official `Notification` layout. The `ProgressBar` uses `@android:style/Widget.Material.ProgressBar.Horizontal`, also mirroring what you will see in real notifications.

Using the Layout

The new sample's `buildForeground()` method now creates a `RemoteViews` for this layout, fills in the title, and uses that with `NotificationCompat.Builder` and `setContent()`:

```
  private NotificationCompat.Builder buildForeground(
    String filename) {
    NotificationCompat.Builder b=
      new NotificationCompat.Builder(this, CHANNEL_WHATEVER);
    RemoteViews content=new RemoteViews(getPackageName(),
      R.layout.notif_content);

    content.setTextViewText(android.R.id.title, "Downloading: "+filename);

    b.setOngoing(true)
      .setContent(content)
      .setSmallIcon(android.R.drawable.stat_sys_download);

    return(b);
  }
```

(from Notifications/CustomView/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

However, `NotificationCompat.Builder` has a write-only API. We cannot get our `RemoteViews` back from that. But, we need the `RemoteViews` to be able to update our progress. So, to that end, we hold onto the actual `Notification` built by the `Builder` in `onHandleIntent()`:

```
      NotificationCompat.Builder builder=
        buildForeground(filename);
      final Notification notif=builder.build();
```

(from Notifications/CustomView/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

Then, in our `ProgressResponseBody.Listener`, we get at the `RemoteViews` via the `contentView` `public` field on the `Notification`. We can call a `setProgressBar()` method on the `RemoteViews`, much as we called `setProgress()` on the `NotificationCompat.Builder` in the preceding example:

```
      final ProgressResponseBody.Listener progressListener=
        new ProgressResponseBody.Listener() {
          long lastUpdateTime=0L;

          @Override
          public void onProgressChange(long bytesRead,
                                       long contentLength,
                                       boolean done) {
            long now=SystemClock.uptimeMillis();

            if (now-lastUpdateTime>1000) {
              notif
                .contentView
                .setProgressBar(android.R.id.progress,
                  (int)contentLength, (int)bytesRead, false);
              mgr.notify(FOREGROUND_ID, notif);
              lastUpdateTime=now;
            }
          }
        };
```

(from Notifications/CustomView/app/src/main/java/com/commonsware/android/okhttp3/progress/Downloader.java)

Then, we can `notify()` the `NotificationManager` with the updated `Notification`, causing the `ProgressBar` to advance based on the actual progress made.

The resulting `Notification` shows our `RemoteViews` in action:

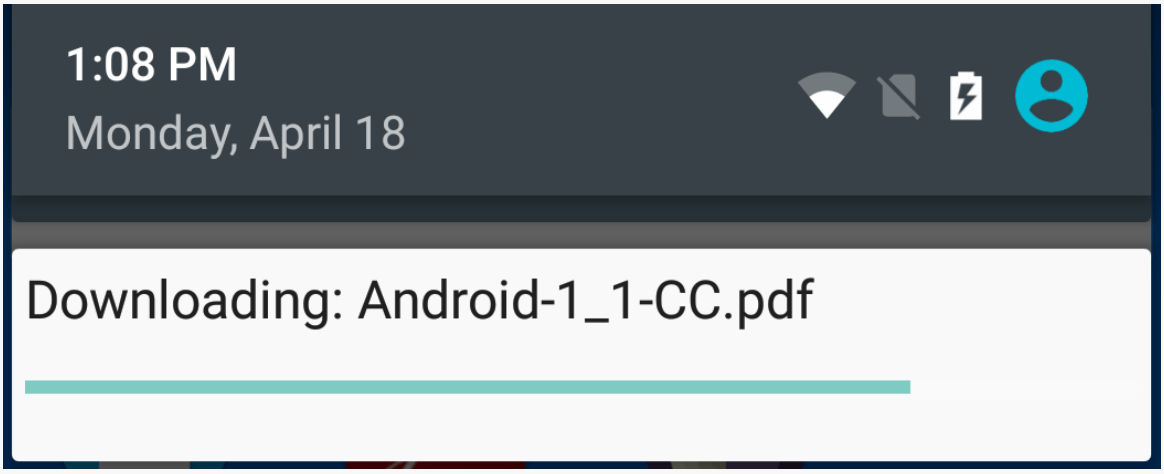


Figure 430: Custom Notification

Styling Custom Views

A custom view for a `Notification` takes over the entire tile in the notification shade. Sometimes, this may be necessary to achieve the developer's objective. Other times, though, while the main content area of the `Notification` might need to be custom, the rest of the "frame" around that content area could be left intact. This would include things like the app's icon, the time the `Notification` was raised, any action buttons below the content, and so forth.

Android 7.0+ offers this via the `Notification.DecoratedCustomViewStyle` and `Notification.DecoratedMediaCustomViewStyle` styles. On your `Builder`, call `setCustomContentView()` with the `RemoteViews` for the content area, plus call `setStyle()`, passing in an instance of `DecoratedCustomViewStyle` or `DecoratedMediaCustomViewStyle`, to gain this effect.

Life After Delete

Most of the time, you do not care about your `Notification` being dismissed by the user from the notification drawer (e.g., pressing the Clear button on Android 1.x/2.x devices). If you do care about the `Notification` being deleted this way, you can supply a `PendingIntent` in the `deleteIntent` data member of the `Notification` — this will be executed when the user gets rid of your `Notification`. Usually, this will be a `getService()` or `getBroadcast()` `PendingIntent`, to have you do something in

the background related to the dismissal. Users are likely to get rather irritated with you if you pop up an activity because they got rid of your `Notification`.

Note that this only works for `Notification` objects that can be cleared. If you have `FLAG_ONGOING_EVENT` set on the `Notification`, it will remain on-screen until *you* get rid of it.

The Mysterious Case of the Missing Number

The `Notification` class has a `number` data member. On Android 1.x and 2.x, setting that data member would cause a number to be super-imposed on top of your icon in the status bar. That data member no longer works as of Android 3.0.

However, `NotificationCompat.Builder` has a `setNumber()` method which *does* work on API Level 11 and higher, though with slightly different behavior. Instead of putting the number on top of your status bar icon, the number will appear in your notification drawer entry. This only works if you do not use `setContent()` with `NotificationCompat.Builder` to define your own notification drawer entry layout — in that case, you could put your own number in wherever you would like.

Notifications and MessagingStyle

Android 7.0 offers a new `MessagingStyle` to the roster of expanded `Notification` styles. This one is designed for a chat-style presentation, where you supply a series of chat messages (person, timestamp, and message), and they are rendered in the `Notification`. It is designed to be used with the `RemoteInput` option described earlier in this chapter, for the user to be able to participate in a chat without having to open up your activity.

As usual, there are two implementations of `MessagingStyle`:

- `Notification.MessagingStyle` is part of Android 7.0's SDK and requires you to build for Android 7.0 (e.g., `compileSdkVersion 24`)
- `NotificationCompat.MessagingStyle`, from the Android Support libraries, for backwards compatibility

The `Notifications/Messaging` sample project demonstrates the use of the latter, along with the `RemoteInput` support from earlier.

While the RemoteInput and MessagingStyle from NotificationCompat will build and run on older devices, they do not work especially well. You simply get a do-nothing Notification action for the RemoteInput and no real context around the messages. As such, you only want to use these options on Android 7.0 devices, gracefully degrading to some other experience on older devices.

Also note that the NotificationCompat.MessagingStyle is from the v24 generation of the Android Support libraries. In this case, we are using 25.0.0 of support-compat, which contains NotificationCompat and NotificationManagerCompat.

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.android.support:support-compat:26.1.0'
}

android {
    compileSdkVersion 26
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
    }
}
```

(from Notifications/Messaging/app/build.gradle)

As with the RemoteInput sample, the MainActivity is here just to provide us with an easy way to get the Notification to appear on the screen. In this case, though, all we do is use NotificationManagerCompat to show a Notification built elsewhere:

```
package com.commonsware.android.messaging;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.app.NotificationManagerCompat;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    NotificationManagerCompat
      .from(this)
      .notify(RemoteInputReceiver.NOTIFY_ID,
        RemoteInputReceiver.buildNotification(this).build());

    finish();
  }
}
```

(from Notifications/Messaging/app/src/main/java/com/commonsware/android/messaging/MainActivity.java)

Our `RemoteInputReceiver` is more complex as a result:

```
package com.commonsware.android.messaging;

import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.RemoteInput;
import android.util.Log;
import java.util.Stack;

public class RemoteInputReceiver extends BroadcastReceiver {
  private static final String CHANNEL_WHATEVER="channel_whatever";
  static final int NOTIFY_ID=1337;
  static final String EXTRA_INPUT="input";
  static final Stack<Message> MESSAGES=new Stack<>();
  static final long INITIAL_TIMESTAMP=System.currentTimeMillis();

  static NotificationCompat.Builder buildNotification(Context ctxt) {
    NotificationManager mgr=
      (NotificationManager)ctxt.getSystemService(Context.NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    Intent i=new Intent(ctxt, RemoteInputReceiver.class);
    PendingIntent pi=
      PendingIntent.getBroadcast(ctxt, 0, i, PendingIntent.FLAG_UPDATE_CURRENT);

    RemoteInput remoteInput=
      new RemoteInput.Builder(RemoteInputReceiver.EXTRA_INPUT)
        .setLabel(ctxt.getString(R.string.talk))
        .build();

    NotificationCompat.Action remoteAction=
      new NotificationCompat.Action.Builder(
```

```
        android.R.drawable.ic_btn_speak_now,
        ctxt.getString(R.string.talk),
        pi).addRemoteInput(remoteInput).build();

    NotificationCompat.MessagingStyle style=
      new NotificationCompat.MessagingStyle("Me")
        .setConversationTitle("A Fake Chat");

    style.addMessage("Want to chat?", INITIAL_TIMESTAMP, "Somebody");

    for (Message msg : MESSAGES) {
      style.addMessage(msg.text, msg.timestamp,
        style.getUserDisplayName());
    }

    NotificationCompat.Builder builder=
      new NotificationCompat.Builder(ctxt, CHANNEL_WHATEVER)
        .setSmallIcon(
          android.R.drawable.stat_sys_download_done)
        .setContentTitle(ctxt.getString(R.string.title))
        .setStyle(style)
        .addAction(remoteAction);

    return(builder);
  }

  @Override
  public void onReceive(Context ctxt, Intent i) {
    Bundle input=RemoteInput.getResultsFromIntent(i);

    if (input!=null) {
      CharSequence text=input.getCharSequence(EXTRA_INPUT);

      if (text!=null) {
        MESSAGES.push(new Message(text));
      }
      else {
        Log.e(getClass().getSimpleName(), "No voice response speech");
      }
    }
    else {
      Log.e(getClass().getSimpleName(), "No voice response Bundle");
    }

    NotificationManagerCompat
      .from(ctxt)
      .notify(RemoteInputReceiver.NOTIFY_ID,
        buildNotification(ctxt).build());
  }

  private static class Message {
    final CharSequence text;
    final long timestamp;

    Message(CharSequence text) {
      this.text=text;
      timestamp=System.currentTimeMillis();
    }
  }
}
```

(from Notifications/Messaging/app/src/main/java/com/commonsware/android/messaging/RemoteInputReceiver.java)

For the `MessagingStyle`, we need messages. In a production app, this would be part of your app's data model, probably saved in a file or database somewhere, with an in-memory cache for speed. In this sample app, we just have a static `MESSAGES Stack`, for our messages. Initially, this `Stack` is empty, but we will eventually fill in `Message` objects, each of which has text and a timestamp. Since this is a sample app, and all `Message` objects will come from our app's user, we do not need `Message` to track the sender of the message — a real chat-style app would need this, in all likelihood.

The `buildNotification()` method starts off with the same basic code shown in the `RemoteInput` sample. But then, mid-way through the method, we build up the `MessagingStyle`:

```
    NotificationCompat.MessagingStyle style=
      new NotificationCompat.MessagingStyle("Me")
        .setConversationTitle("A Fake Chat");

    style.addMessage("Want to chat?", INITIAL_TIMESTAMP, "Somebody");

    for (Message msg : MESSAGES) {
      style.addMessage(msg.text, msg.timestamp,
        style.getUserDisplayName());
    }
```

(from Notifications/Messaging/app/src/main/java/com/commonsware/android/messaging/RemoteInputReceiver.java)

The parameter to the `MessagingStyle` constructor is the name associated with the user of this app. That name will appear alongside messages that come from this user. A `MessagingStyle` can have a title, set via `setConversationTitle()`, to provide some context for the chat transcript.

We then add one fake message, ostensibly from the chat partner, via the `addMessage()` method. The version we use here takes the text, timestamp, and name of the other party to use in the message.

Then, if there are messages in the `MESSAGES` stack, we add those to the chat transcript as well. Note that our third parameter is `style.getUserDisplayName()`, which returns the value that we passed into the `MessagingStyle` constructor. We could get the same effect by passing `null` for the third parameter.

That `MessagingStyle` then is attached to the `NotificationCompat.Builder` via `setStyle()`.

At the outset, since `MESSAGES` is empty, we get a single message from "Somebody":

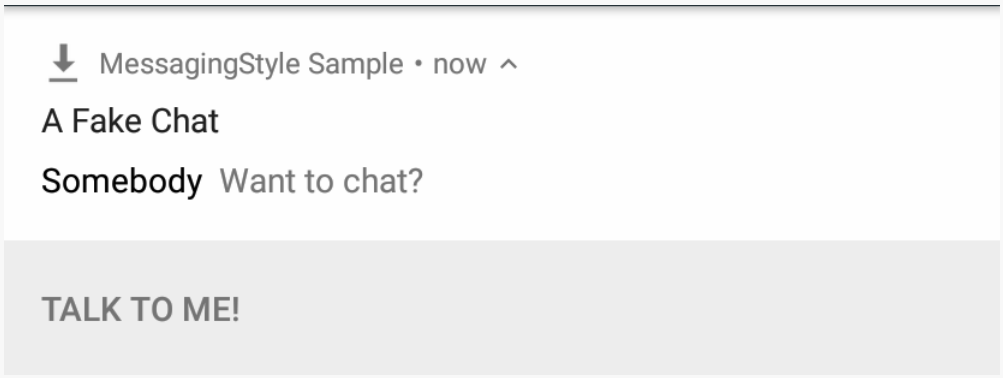


Figure 431: Messaging Demo Notification, As Initially Launched

If the user taps "Talk to Me", types in a message, and clicks the send button, our `RemoteInputReceiver` will take that text, put it in a `Message`, push that `Message` into the `MESSAGES` `Stack`, then update the `Notification` with the new transcript:

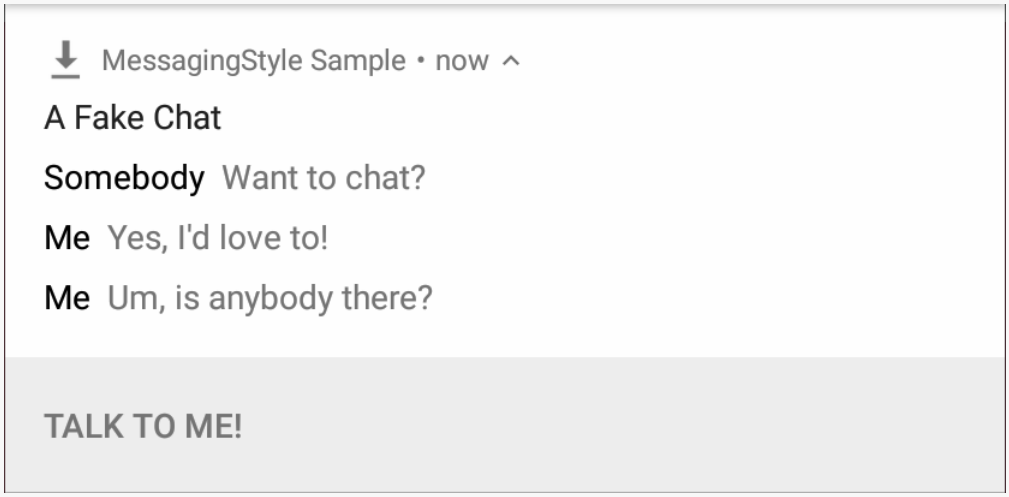


Figure 432: Messaging Demo Notification, After Two Replies

The timestamp does not show up; it is unclear how that is used by Android.

Changes in API Level 23

Historically, while we could supply a "large bitmap" (e.g., photo or avatar) to a `Notification` for use in the tile in the notification tray as a `Bitmap`, the "small icon"

used for the status bar always had to be a resource in our app. This was aggravating for developers that wanted to tailor the small icon, such as a weather app showing the current temperature. Now, we can supply an `Icon` object, which can wrap a drawable resource, a `Uri` to a `ContentProvider`, a `byte` array of encoded bitmap data, a `Bitmap`, or a path to a local PNG or JPEG file. Any of those can be used for the small icon, offering greater flexibility. That being said, please do bear in mind that the small icon is small (i.e., tiny changes may not be noticeable) and that ideally it should adhere to the platform aesthetic for notification icons (i.e., do not use a photo).

The user can now disable our heads-up notifications, if the user finds them irritating. We can get an idea of what the user's chosen notification policies are via `getCurrentInterruptionFilter()` and `getNotificationPolicy()`, so we have some general sense of what the user is and is not expecting to see in terms of notifications.

And, at long last, we can find out all of our active notifications, via a `getActiveNotifications()` method. This will include any notification that is visible to the user (i.e., the user has not dismissed it and we have not gotten rid of it via `cancel()`).

Sounds and Android 7.0

You can put a custom ringtone on a `Notification`, via methods like `setSound()` on `NotificationCompat.Builder`. This requires a `Uri`. And, unfortunately, your options for that `Uri` are limited:

- `android.resource` `Uri` values are fine, if the sound that you want to play is a raw resource
- `file` `Uri` values will work prior to Android 7.0 and for apps whose `targetSdkVersion` is below 24
- `content` `Uri` values are the preferred solution for sounds that exist as files (e.g., by using `FileProvider`), but they do not work on Android 7.0 without additional work, because Android may not have read access to your content

The best easy solution at the moment, for `content` `Uri` values, is to grant read access to your content to the `com.android.systemui` package, via the `grantUriPermissions()` method on `Context`:

```
grantUriPermission("com.android.systemui", sound,
  Intent.FLAG_GRANT_READ_URI_PERMISSION);
```

(where `sound` is the `Uri` that you want to use with `setSound()`)

However, it is unclear if this is the right package to use for all versions of Android and all possible device manufacturer modifications to Android.

Another possibility is for you to create a read-only `ContentProvider` that can serve your file, perhaps modeled after this book sample, then export that provider. The `FileProvider` from the Android Support library would work, except that it cannot be exported.

Eventually, the author's `StreamProvider` will be updated to support this sort of public read-only pattern.

With luck, future versions of Android (and the `NotificationCompat` backport) will address this more formally.

Auto-Timeout

In Android 8.0+ can now provide a time when the notification should be automatically deleted, as if the user had cleared it personally. This is good for notifications whose value declines a lot after a particular point in time.

To use this, call `setTimeoutAfter()` on the `NotificationCompat.Builder`, supplying a duration in milliseconds:

```
public void raiseCoins(View view) {
  Notification n=new NotificationCompat.Builder(MainActivity.this, CHANNEL_COINS)
    .setContentTitle(getString(R.string.notif_coins_title))
    .setContentText(getString(R.string.notif_coins_text))
    .setSmallIcon(android.R.drawable.stat_sys_warning)
    .setTimeoutAfter(5000)
    .build();

  mgr.notify(NOTIF_ID_COINS, n);
```

(from Notifications/Channels/app/src/main/java/com/commonsware/android/notify/channel/MainActivity.java)

Here, we set the notification to time out after five seconds. Five seconds would be far too short of an active window for ordinary users, but it allows a developer to see the results more quickly.

Colorized Notifications

Calling both `setColorized(true)` and `setColor()` (with a specific color) on a `NotificationComapat.Builder` indicates that you want the color to be used as a highlight color. For a regular notification, this is used to colorize the title. For other notification styles, this may be used to set the background color. Since you have no control over exactly where this color gets used, using this feature is not a particularly good idea. Also, it will only have a visible effect on Android 8.0+.

Launcher Icon Badge

Depending on the home screen launcher implementation, your notifications *may* display a dot on the home screen launcher icon:

Figure 433: Home Screen Launcher Notification Dot, from Pixel

This will happen by default; you do not need to do anything to make it happen. However, it will only appear on a handful of devices that have a compatible home screen.

You can disable this by calling `setShowBadge(false)` on the `NotificationChannel` when you create it.

For launchers that support badges, a long-press on the launcher icon may show details of the notification:

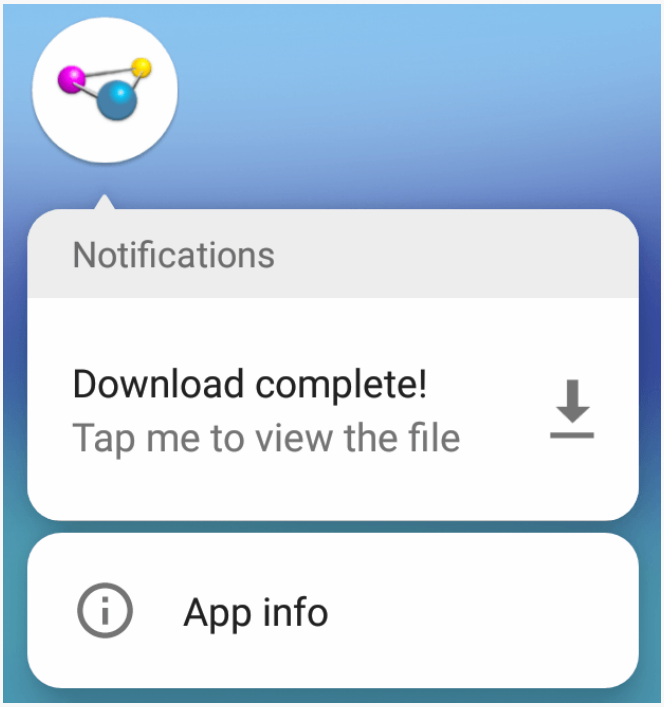


Figure 434: Home Screen Launcher Notification Details, from Pixel

You can control two elements of this, beyond what you would normally configure in the notification itself:

- Call `setBadgeIconType()` on the `NotificationCompat.Builder` to indicate whether the icon should be displayed or not, and whether the small or the large icon will be used
- Call `setNumber()` on the `NotificationCompat.Builder` to offer up a number to show in the details:

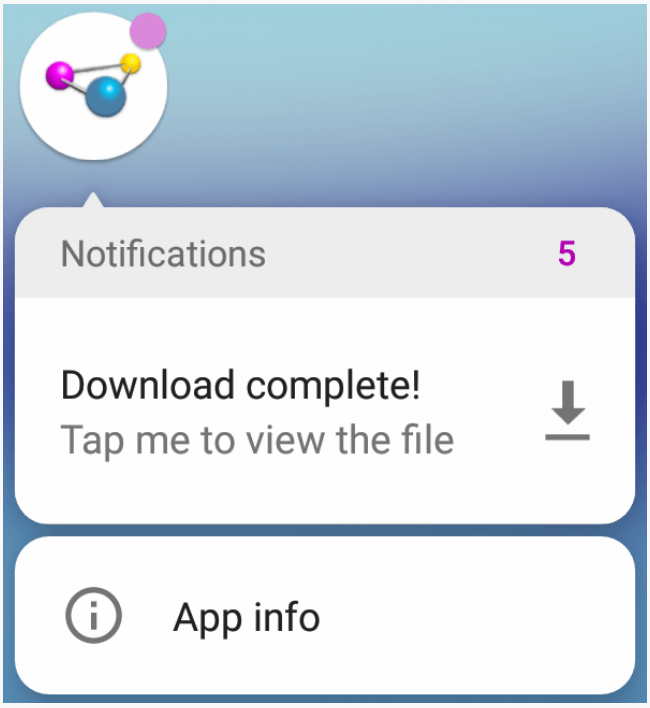


Figure 435: Home Screen Launcher Notification Details With Number, from Pixel

For example, you might use this for an unread message count, if that information is not otherwise present in the notification.

Multi-Window Support

From the standpoint of users, the most visible new capability in Android 7.0 is multi-window support. Now, the user can be working with more than one Android activity at a time, whether from separate apps or — with the assistance of the app — from the same app. This is akin to the proprietary implementations seen in devices from Samsung, LG, and other manufacturers.

The good news — more or less — is that support for multi-window is automatic. You do not need to change anything in your Android app to have your app moved into a portion of the screen, instead of taking up the full screen.

However, you may want to tweak your app to behave better in a multi-window environment.

Prerequisites

Understanding this chapter requires you to have read the core chapters of the book.

A History of Windows

In the beginning, we were happy to have just one app on the screen at a time.

However, as phones got larger, there was increasing interest in having more than one app visible at a time. Some manufacturers handled this via their own “small apps”, such as floating calculator windows.

However, some, such as Samsung and LG, added support for split-screen dual-window environments, where two apps could be run side-by-side. These were proprietary extensions to Android that developers had to opt into.

Other manufacturers, such as Jide, set up alternative versions of Android (e.g., Jide's Remix OS) that supported windows akin to desktop operating systems, with an arbitrary number of overlapping windows. These too were proprietary extensions, unique to those environments.

With Android 7.0, these sorts of capabilities are now part of the core OS, with standardized ways for developers to work with them.

What The User Sees

For most Android devices, the user experience will be what is known as the split-screen view:

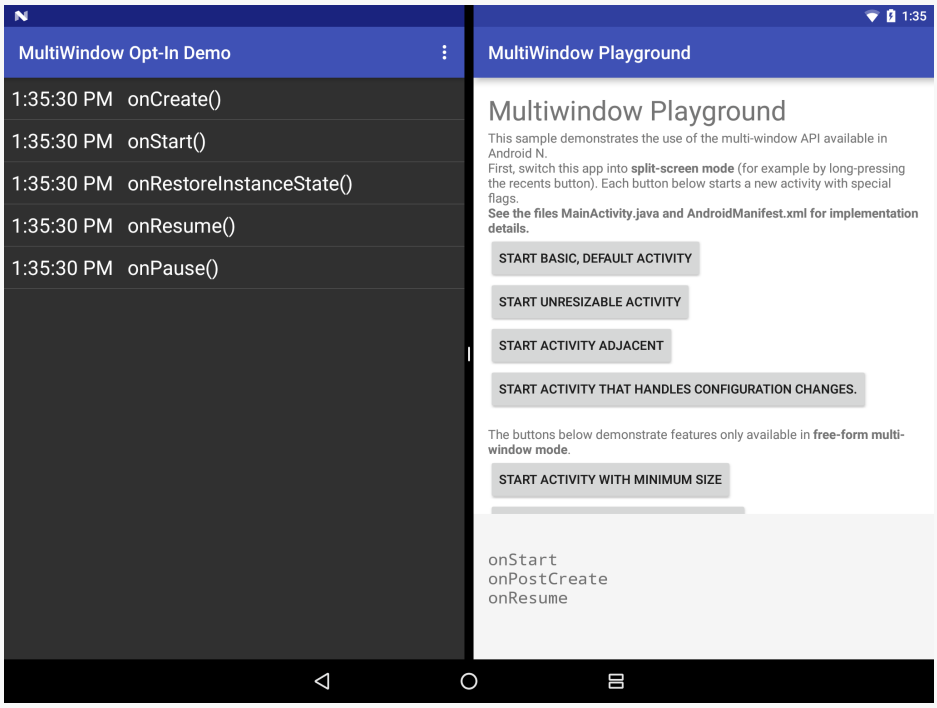


Figure 436: Split-Screen Mode on Nexus 9

The user can enter split-screen mode by long-pressing on the OVERVIEW button (the one that brings up the recent tasks). The existing foreground activity will be put

in one pane, with the overview screen in the other pane, where the user can choose another app.

If the user rotates the screen, the split-screen remains, still splitting along the long axis:

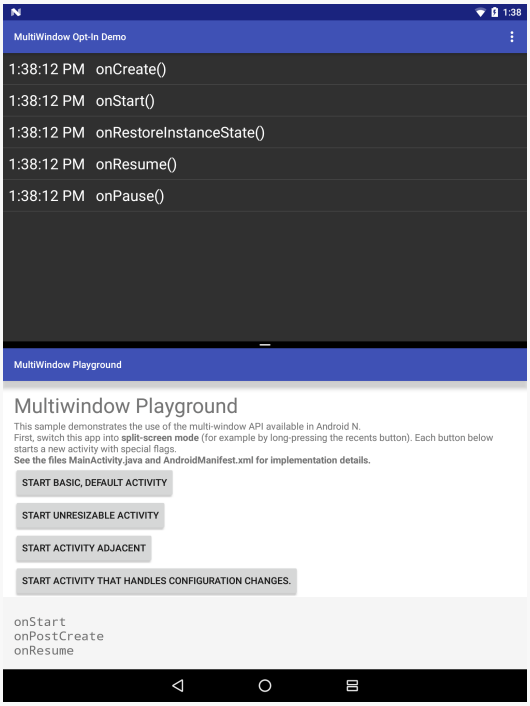


Figure 437: Split-Screen Mode on Nexus 9, Portrait Mode

The divider is movable between three positions, to either equally split the space (default) or to give one pane or the other about two-thirds of the space:

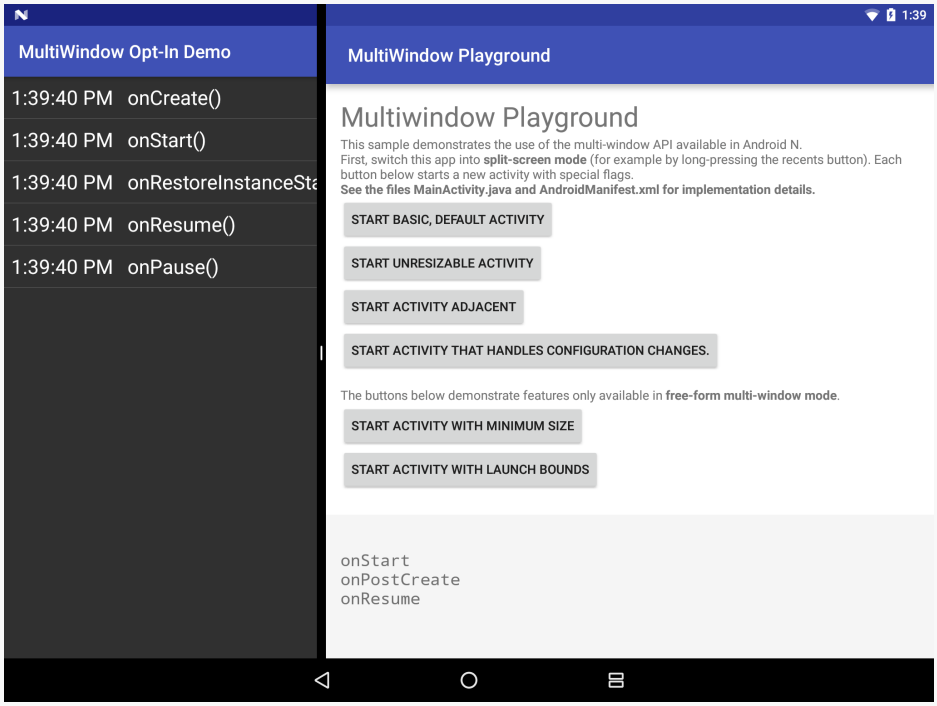


Figure 438: Split-Screen Mode on Nexus 9, After Moving Divider

Android TV devices support a "picture-in-picture" mode instead, where one activity is in a small floating window, overlaying the other activity.

The documentation also describes a "freeform" mode, where Android behaves like a desktop OS with overlapping fully-resizable windows. At the time this paragraph was written (October 2016), no production devices officially support freeform multi-window, though power users have a way of enabling it.

What Your Code Sees

From your activity's standpoint, the fact that it once used most of the screen, and now is only smaller part of a screen, is just a configuration change, no different than orientation changes or other screen size changes (e.g., putting the device in a dock that provides a larger screen).

Whatever activity the user tapped on last is considered to be the foreground activity. Other activities that are visible, such as the activity in the adjacent split-screen pane, will be paused. This is in line with the way Android has always worked:

- if your activity is visible, but not in the foreground, it will be paused (e.g., a system-supplied dialog-themed activity is in the foreground)
- if your activity is no longer visible, it will be stopped

You will want to think through what business logic of your activities belongs in `onStart()`/`onStop()` and what belongs in `onResume()`/`onPause()`. Historically, since being visible but not in the foreground was an uncommon, short-lived state, we did not necessarily have to worry that much about the distinction between "paused" and "stopped". Now the distinction takes on much greater importance.

In split-screen mode, if the user moves the divider, your activity initially will be simply redrawn to adopt the extra space. Once the user lets go of the divider, and it settles on its final position, your activity may undergo a configuration change. Whether you undergo a configuration change seems somewhat random. With luck, this will be more predictable in the future.

If needed, activities and fragments can find out what is going on with respect to multi-window behavior:

- They can call `isInMultiWindowMode()` to find out if they are in multi-window mode presently... in theory
- They can override `onMultiWindowModeChanged()` to find out if the multi-window state changes

There are also variants of these for picture-in-picture mode for Android TV: `isInPictureInPictureMode()` and `onPictureInPictureModeChanged()`. However, picture-in-picture mode is a particular case of multi-window mode. For example, if `isInPictureInPictureMode()` returns `true`, so will `isInMultiWindowMode()`.

However, `isInMultiWindowMode()` is unreliable, apparently by design. Ideally, avoid doing anything specific for when you are in multi-window mode or not.

Also note that, as of Android 8.0, there are two variants of `onMultiWindowModeChanged()`:

- A one-parameter version, taking a `boolean`, which was the original method added in Android 7.0 but is marked as deprecated in Android 8.0
- A two-parameter version, taking a `boolean` and a `Configuration`, added in Android 8.0

In general, if your `minSdkVersion` is 26 or higher, override the two-parameter version. Otherwise, override the one-parameter version.

Opting Out

For various reasons, you may not want your activity to be eligible to be used in some form of multi-window mode. This may disappoint your users, but you may have valid reasons for this decision.

If your `targetSdkVersion` is `N` (or whatever that turns into, probably `24`, when Android 7.0 ships in final form), you can have an `android:resizeableActivity="false"` attribute on a specific `<activity>` element or on the `<application>` element in your manifest. This will tell Android to always give you the full screen, even if the user tries launch your activity into some form of multi-window mode.

If your `targetSdkVersion` is `23` or lower, whether you support multi-window mode is determined by the `android:screenOrientation` attribute (on an `<activity>` or inherited from the `<application>`). A fixed-orientation activity — such as one that is locked to landscape – will not be put into multi-window mode.

Note that if your `targetSdkVersion` is `23` or lower, and you support any orientation (e.g., you do not have `android:screenOrientation`), Android will allow the user to use your activity in multi-window mode. However, a `Toast` will appear, advising the user that your activity is not designed for multi-window mode and there may be compatibility issues. However, this will serve as a note to users that your app is out of date with respect to newer versions of Android, which may not be in your best interests.

Opting In

As noted, Android will allow your activity to be put in multi-window mode by default.

To avoid the aforementioned warning `Toast`, set your `targetSdkVersion` to `N`. Optionally, you can explicitly have `android:resizeableActivity="true"` in the manifest, though this is the default.

Note that not all activities need to support multi-window mode. For example, you might have some general activities that are fine in multi-window mode, plus a video

player that really should be full-screen in landscape. You might put `android:resizeableActivity="false"` on that latter activity, plus have `android:resizeableActivity="true"` on the `<application>` element (for documentation purposes).

If, for some reason, regular multi-window modes are fine, but picture-in-picture will be a problem, you can have `android:supportsPictureInPicture="false"` on an `<activity>`.

Configuring the Layout

You can add a `<layout>` element as a child to your `<activity>` element, to control aspects of how the activity appears on the screen in multi-window mode.

The only facet of `<layout>` that we can use today is the minimal size, represented by `android:minWidth` and `android:minHeight` attributes. These indicates how small you are willing to have your activity be in the stated direction. If the user moves the divider, and the resulting size of your activity is smaller than your requested minimal size, your activity will appear to extend "under" the other pane.

For example, the `MultiWindow/OptIn` sample application has its minimal width set to 3 inches (`480dp`):

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:resizeableActivity="true"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name="MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <layout android:minWidth="480dp" />
    </activity>
  </application>

</manifest>
```

(from MultiWindow/OptIn/app/src/main/AndroidManifest.xml)

When positioned on the left of the split-screen, with a size less than 3 inches, the activity "extends beneath" the right pane:

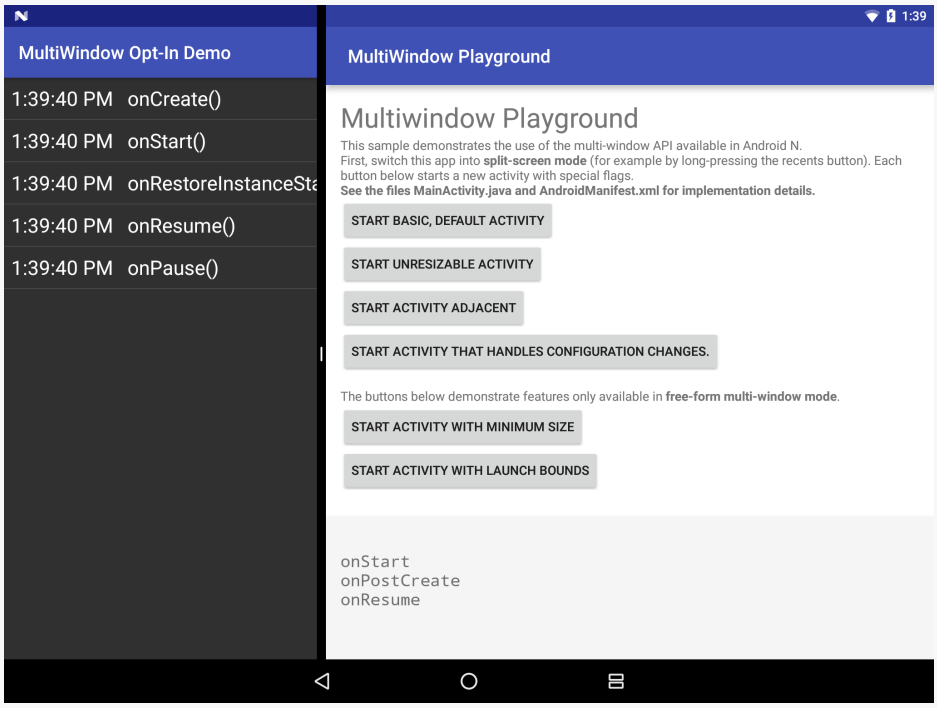


Figure 439: 3-Inch Minimum Size on Left of Split-Screen

The `OptIn` sample app has a `ListView` showing the events received in the lifetime of this activity instance, so you can see the effect of tapping on one activity versus the other in the split-screen mode.

Also, the other activity shown in these screenshots is Google's official multi-window sample app.

For freeform multi-window mode, you also have:

- `android:defaultWidth` and `android:defaultHeight`, to supply a suggested size for your window (as a dimension or as a fraction of the screen size), and
- `android:gravity`, which works like the equivalent widget attribute, suggesting where on the screen your window should be opened

Avoiding Stutter

Since resizing an activity in multi-window mode may cause a configuration change, it is very important for you to handle configuration changes quickly:

- For small bits of data that can be put in a `Bundle`, use the saved instance state `Bundle`, so that your new activity (and fragments) can not only rapidly handle these configuration changes but also handle other scenarios, such as your process being terminated while in the background.
- For larger chunks of data, or data that cannot be put into a `Bundle`, use a retained fragment or a process-level cache. Be careful with the latter technique, though, so that you do not consume too much heap space.

What you want to avoid, if at all possible, is having to do I/O of any form due to a configuration change.

In addition, while the user is resizing your activity, it is simply being redrawn using its current UI, pending completion of the resize (at which point, a configuration change may occur). The more work you do to render the UI, the more work that needs to be done to redraw the UI while the user is resizing, and the more likely that it is that the user will perceive some jank. Possible problem areas include:

- Having hundreds of widgets in your activity
- Having custom widgets that are expensive to redraw
- Triggering some sort of controller-style logic due to a redraw that in turn triggers more serious work (e.g., "we want to log every time the widget gets drawn to a file")

In a pinch, you can optimize certain configuration changes, overriding the default activity destroy-and-recreate cycle, via `android:configChanges`. On the plus side, you can try to make fine-grained changes to your UI and react more responsively. However, *this is an optimization*, not a replacement for proper state management (e.g., saved instance state `Bundle`), as state management is for more than configuration changes.

If you wish to use `android:configChanges` to opt out of automatic handling of certain configuration changes, the ones of relevance for multi-window are:

- `screenSize`
- `smallestScreenSize`

- `screenLayout`
- `orientation`

If you have not used this technique before, you can read more about it in the chapter on configuration changes.

Managing the Background

While the user is resizing the window, Android does not attempt to re-render your UI. Instead, if the window is being shrunk, your existing UI is clipped. If the window is being expanded, a background is shown over the new area. Only once the resize is done does Android perform the configuration change and re-render your UI.

The `android:windowBackground` and `android:windowBackgroundFallback` theme attributes control what that background looks like. You may wish to set `android:windowBackground` in your app's theme to a value that matches your natural window background, so there is a seamless transition between your regular background and the new background added by Android during the resize operation.

How Low Can You Go?

The smallest window size in split-screen mode is 220dp. Your activity should aim to support a width or height of 220dp for maximum compatibility. Using `android:minWidth` and `android:minHeight` will allow your activity to support those small sizes by having your UI be clipped, but this is not an ideal user experience. Rely on `android:minWidth` and `android:minHeight` only for cases where you have no good way of supporting 220dp directly.

It is unclear whether this 220dp minimum also holds for freeform or picture-in-picture multi-window mode.

Handling the Screen Size Transition

Suppose your activity launches in a window size that, based on your layout rules, pulls in a phone-sized layout resource. Now, the user resizes your window, and the resulting size would pull in a tablet-sized layout resource.

Assuming that you are handling all of this properly via the configuration change, technically your activity should work just fine. But from the user's standpoint, it may

result in a jarring transition, if the UI for one screen size is significantly different from the UI for another screen size.

You sometimes see this with Web sites. Some sites apply their site designs based solely on the size of the browser viewport, so they are not dependent upon flaky ways of detecting whether the browser is coming from a mobile device or not. If the viewport is small enough, the page's CSS renders a mobile-friendly UI; larger viewports result in more of a desktop feel. However, if this is based on CSS, resizing a desktop browser window to be small causes the Web page to dynamically shift from desktop to mobile mode, or vice versa. This puts some stress on the Web page design, so that the design not only works statically (i.e., a small rendition works well on mobile) but also dynamically (i.e., the user does not get too confused when the Web page transitions from one set of CSS rules to another).

We will wind up with the same problem in multi-window on Android, as the user resizes windows past our natural transition points.

Ideally, your app uses one UI for everything from small phones (or small windows in multi-window) to large tablets (or large windows in multi-window), regardless of window orientation. Few UI designs work well this way. And since you cannot reliably determine whether or not you are in multi-window mode via `isInMultiWindowMode()`, you cannot reliably treat that as a separate case.

As such, the community will eventually need to evolve some patterns for handling this scenario.

Parallel Processing

Normally, multi-window is for multiple apps. For example, the user might be watching a video in one pane while taking notes in another.

However, there will be cases where it might help the user to have two activities of yours be in the panes of the split-screen mode. Or, there may be cases where users want to launch some content of yours into a separate window in freeform mode.

To do this, you can add `FLAG_ACTIVITY_LAUNCH_ADJACENT` to the `Intent` that starts up another activity. If the device is in some form of multi-window mode, this serves as a hint that you want this new activity to be in a different pane or window than is the current activity. If the device is *not* in multi-window mode, adding this flag has no effect — you cannot force a device into multi-window mode.

Because this is only conditionally available, you will want to set up your UI to reflect that fact. Possible strategies include:

- Only offering a "start in a new pane/window" option if `isInMultiWindowMode()` returns `true`, or toggling its availability in `onMultiWindowChanged()`
- Always having the option to start the activity in a new pane or window, but if `isInMultiWindowMode()` returns `false` at that point, show a dialog or `Snackbar` or something to point out that the user has to set up multi-window mode first

However, the two activities (e.g., the ones in each pane of split-screen) need to be part of separate tasks. The recipe for doing this is to not only use `FLAG_ACTIVITY_LAUNCH_ADJACENT`, but also `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_MULTIPLE_TASK`.

For example, `MainActivity` in the `OptIn` sample app has an overflow menu with a "Clone" menu item. This opens a second instance of `MainActivity` into the other split-screen pane:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.clone) {
    Intent i=
      new Intent(this, MainActivity.class)
        .setFlags(Intent.FLAG_ACTIVITY_LAUNCH_ADJACENT |
          Intent.FLAG_ACTIVITY_NEW_TASK |
          Intent.FLAG_ACTIVITY_MULTIPLE_TASK);

    startActivity(i);
    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from MultiWindow/OptIn/app/src/main/java/com/commonsware/android/multiwindow/MainActivity.java)

Note that the `Intent` gets all three of the aforementioned flags: `FLAG_ACTIVITY_LAUNCH_ADJACENT`, `FLAG_ACTIVITY_NEW_TASK`, and `FLAG_ACTIVITY_MULTIPLE_TASK`.

The chapter on consuming documents using the Storage Access Framework profiles a `TinyTextEditor` sample app that further demonstrates how to open a separate

window from an existing one. In this case, it will be to move a text editor from a tab in one activity into a separate activity in its own window.

You also have the option of using the two-parameter `startActivity()` that takes a `Bundle`, building that `Bundle` using the `ActivityOptions` class. On `ActivityOptions`, there is `setLaunchBounds()`, to indicate where on the screen the new task's window should appear. The parameter to `setLaunchBounds()` is either a `Rect` (providing that size/location in screen coordinates) or `null` (indicating that the new task's window should occupy the full screen).

Split-Screen, HOME, and Your Activity

We are used to the notion that when the user presses HOME, we are called with `onStop()`. In fact, `onStop()` is even more important than before with Android 7.0, as our activities will be paused (but not stopped) a lot more with multi-window. So, whereas we might have used `onResume()` and `onPause()` for setting up and tearing down foreground work, we might now switch to `onStart()` and `onStop()`.

However, `onStop()` is not always called when the user presses HOME.

If the device is in normal, non-split-screen mode, and the user presses HOME, your foreground activity moves to the background, and `onStop()` is called.

If the device is in split-screen mode, and your activity is in the bottom or right pane (depending on device orientation), and the user presses HOME, your activity is stopped, so you get an `onStop()` call. Visually, the activity animates off-screen fully, and if the user presses the RECENTS button, the pane you had been occupying is replaced with the recent-tasks list.

But, if your activity is in the top or left pane of split-screen, and the user presses HOME, **your activity is not stopped**.

Unfortunately, we are not given any sort of callback or other indication that this has occurred. The theory is that the device will be in this "transient home" state for a short period of time, but that is not guaranteed.

For many activities, there is no real problem, but if your activity is holding a wakelock (e.g., you are using `android:keepScreenOn` or `setKeepScreenOn()`), you are not given an opportunity to release that wakelock, so as long as the device stays in this state, you drain the battery.

For something like a wakelock, you could release that in `onPause()`, while continuing to do the "real work" of the activity until `onStop()`. That is not ideal — the user might be intentionally watching the video in split-screen and want the screen to stay on. However, until we get some means of identifying this state, our options are limited.

Split-Screen and Orientations

If an activity of yours is in the foreground, and the user enters into split-screen mode, your activity will undergo a configuration change. We tend to focus on the fact that the amount of screen space is cut in half. However, in addition, your orientation is likely to change, as half the screen probably has the opposite orientation from the full screen.

As a result, if you have resources, such as layouts, that are orientation-dependent, as the user flips between split-screen and normal modes, your activity will change orientations. This may even occur if the user drags the divider between the split-screen panes, depending on the aspect ratio of the device screen.

From a programming standpoint, this should not be a problem, as there are many more common ways for a device to change orientation (e.g., user rotates the device). However, the user might not expect a significant layout change based solely on entering or exiting split-screen mode, whereas they may be more comfortable with such a change when rotating the screen.

Furthermore, freeform multi-window mode allows the user to arbitrarily resize the window. In that case, the user might switch orientations by resizing the window.

Aim to minimize orientation-dependent resources. Where you need them, try to make the transition between orientations fairly gentle, so the user does not have an adverse reaction to seeing your UI shift on the fly.

Forcing Your App Into Multi-Window/Multi-Instance

Android 7.0 started shipping in August 2016. By September 2016, apps were appearing on the Play Store that allow users to do what some bloggers think is launch multiple instances of an app.

In reality, they launch multiple instances of your launcher activity in separate panes of a split-screen multi-window environment. So, for example, you could have two

instances of Google's News and Weather up side-by-side, even though the app does not directly support this:

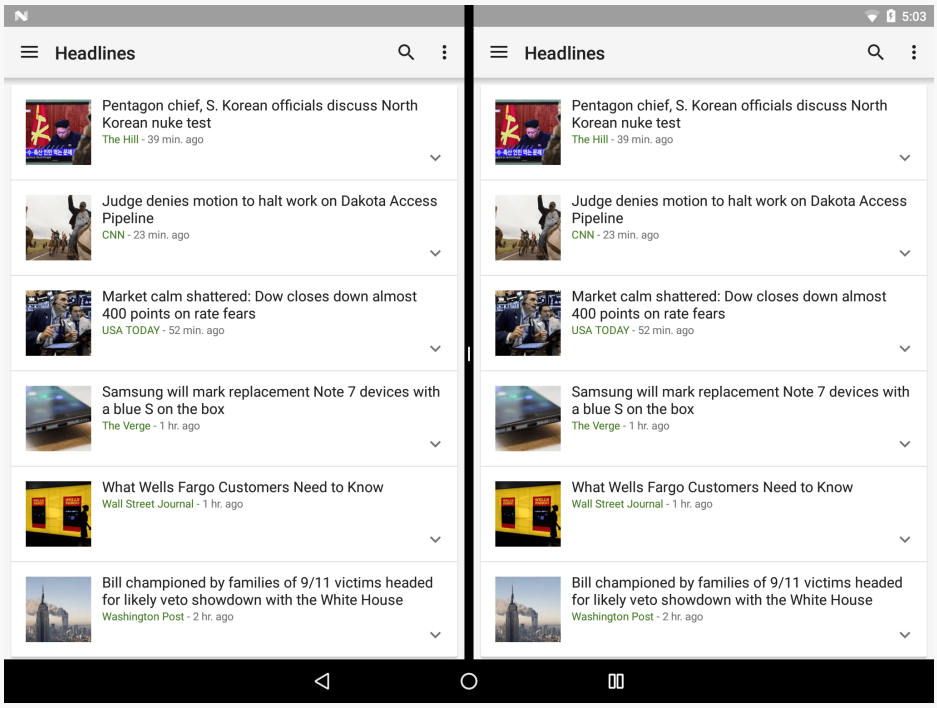


Figure 440: Pair of "News and Weather" Activities

This does not require (much) devious hacking, and it is very possible that we will start seeing device manufacturers offer this sort of capability built into their Android 7.0+ devices.

To understand what's going on, let's examine the `Introspection/Sidecar` sample application. This sample application allows the user to add a custom tile to the quick-settings area of the notification shade. When this Sidecar tile is tapped, a user-specified activity will launch into one of the panes of the split-screen mode. And, as a bonus, if the activity tries blocking split-screen operation via `android:resizeableActivity="false"`, the Sidecar gets around that. The net effect is that any Sidecar user can get two instances of a launcher activity side-by-side... at least, for most apps. For example, the screenshot shown above was set up via Sidecar.

Using Sidecar

Sidecar is not really a production-ready app.

That being said, if you want to play around with this, run and install the app on an Android 7.0+ device. You will be greeted by an activity that shows a list of candidate activities for the Sidecar tile to open. Tap on one. The activity will vanish, showing a "Saved!" `Toast`.

Then, open the notification shade all the way and click "Edit" in the quick-settings area. You should see a "Sidecar" tile that you can drag into the quick-settings area:

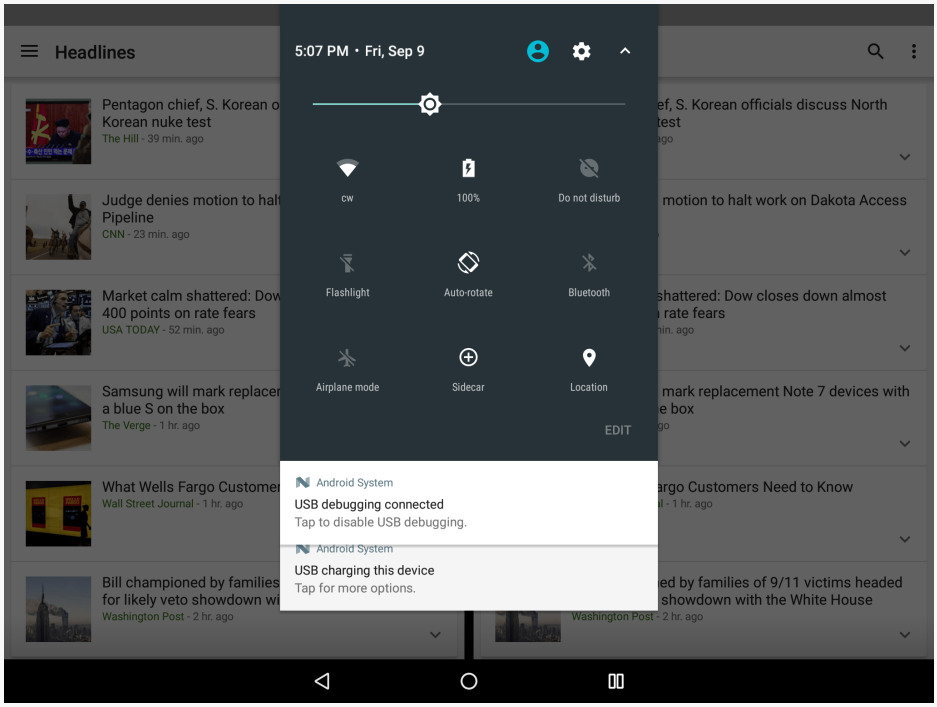


Figure 441: Quick Settings, Showing Sidecar Tile

At this point, if you enter into multi-window mode, then open the notification shade and tap on the Sidecar tile, it will launch the activity you chose into one of the available windows. In mobile device split-screen mode, usually the Sidecar-launched activity will appear in the top or left pane. This will happen even if you have another instance of that same activity in the opposite pane.

Choosing the Activity

Sidecar's `MainActivity` is a near-clone of the `Launchalot` activity profiled elsewhere in the book. It uses `PackageManager` and `queryIntentActivities()` to find all activities that have the `ACTION_MAIN`/`CATEGORY_LAUNCHER` `<intent-filter>`, sorts them alphabetically, and displays them in a `ListView`. The differences here are the

filtering performed in Sidecar and what happens when the user taps on a `ListView` row.

In `Launchalot`, the entire roster of launchable activities is shown in the `ListView`. In Sidecar's `MainActivity`, only a subset are shown, specifically rejecting those whose `android:launchMode` is set to `singleInstance` or `singleTask`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  PackageManager pm=getPackageManager();
  Intent main=new Intent(Intent.ACTION_MAIN, null);

  main.addCategory(Intent.CATEGORY_LAUNCHER);

  List<ResolveInfo> launchables=pm.queryIntentActivities(main, 0);
  List<ResolveInfo> filtered=new ArrayList<>();

  for (ResolveInfo launchable : launchables) {
    int launchMode=launchable.activityInfo.launchMode;

    if (launchMode!=ActivityInfo.LAUNCH_SINGLE_INSTANCE &&
      launchMode!=ActivityInfo.LAUNCH_SINGLE_TASK) {
      filtered.add(launchable);
    }
  }

  Collections.sort(filtered,
                   new ResolveInfo.DisplayNameComparator(pm));

  adapter=new AppAdapter(pm, filtered);
  setListAdapter(adapter);
}
```

(from Introspection/Sidecar/app/src/main/java/com/commonsware/android/sidecar/MainActivity.java)

So, we iterate over the `launchables`, find the `launchMode` for each, and only add the "good" ones to `filtered`. The `filtered` list is what gets shown in the `ListView`.

In `Launchalot`, when the user taps on a row, we create an `Intent` identifying that particular activity, then start up that activity. In Sidecar's `MainActivity`, we save the `ComponentName` identifying the activity to a `SharedPreferences` for later use:

```
@Override
protected void onListItemClick(ListView l, View v,
```

```
                                int position, long id) {
    ResolveInfo launchable=adapter.getItem(position);
    ActivityInfo activity=launchable.activityInfo;
    ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                         activity.name);

    PreferenceManager
      .getDefaultSharedPreferences(this)
      .edit()
      .putString(SidecarTileService.PREF_TO_LAUNCH,
        name.flattenToString())
      .apply();
    Toast
      .makeText(this, R.string.msg_saved, Toast.LENGTH_LONG)
      .show();
    finish();
  }
```

(from Introspection/Sidecar/app/src/main/java/com/commonsware/android/sidecar/MainActivity.java)

The easiest way to persist a `ComponentName` is to use `flattenToString()`, then persist the `String`. That value can later be used with the `unflattenFromString()` `static` method on `ComponentName` to get back an equivalent `ComponentName`, as will be seen shortly.

So, the job of `MainActivity` is to let the user choose an activity for the Sidecar to launch.

Implementing the TileService

A `TileService` is the Android 7.0+ way to set up tiles that the user can add to the quick-settings area of the notification shade. Full details on how to set up one of those can be found elsewhere in the book. Suffice it to say that we are called with `onClick()` in a subclass of `TileService` — named `SidecarTileService` — when the user taps on our tile. At this point, what we want to do is launch the user's requested activity.

```
package com.commonsware.android.sidecar;

import android.content.ComponentName;
import android.content.Intent;
import android.content.SharedPreferences;
import android.content.pm.ActivityInfo;
import android.preference.PreferenceManager;
import android.service.quicksettings.TileService;
```

```
import android.util.Log;
import android.view.Gravity;
import android.widget.Toast;
import java.lang.reflect.Field;

public class SidecarTileService extends TileService {
  static final String PREF_TO_LAUNCH="toLaunch";

  @Override
  public void onClick() {
    super.onClick();

    SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(this);
    String cnFlat=prefs.getString(PREF_TO_LAUNCH, null);

    if (cnFlat!=null) {
      ComponentName cn=ComponentName.unflattenFromString(cnFlat);

      try {
        ActivityInfo info=getPackageManager().getActivityInfo(cn, 0);
        Intent i=new Intent().setComponent(cn);
        Field f=ActivityInfo.class.getField("resizeMode");
        Integer resizeMode=(Integer)f.get(info);
        boolean resizeable=(resizeMode.intValue()!=0);

        if (resizeable) {
          i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
            Intent.FLAG_ACTIVITY_MULTIPLE_TASK |
            Intent.FLAG_ACTIVITY_LAUNCH_ADJACENT);

          startActivity(i);
        }
        else {
          Intent taskRoot=
            new Intent(this, TaskRootActivity.class)
              .putExtra(Intent.EXTRA_INTENT, i)
              .setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
                Intent.FLAG_ACTIVITY_MULTIPLE_TASK |
                Intent.FLAG_ACTIVITY_LAUNCH_ADJACENT);

          startActivity(taskRoot);
        }
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
          "Exception trying to launch activity", e);
```

```
        toast(R.string.msg_sorry);
      }
    }
    else {
      toast(R.string.msg_choose);
    }
  }

  private void toast(int msg) {
    Toast t=Toast.makeText(this, msg, Toast.LENGTH_LONG);

    t.setGravity(Gravity.END | Gravity.BOTTOM, 32, 32);
    t.show();
  }
}
```

(from Introspection/Sidecar/app/src/main/java/com/commonsware/android/sidecar/SidecarTileService.java)

First, we get our `SharedPreferences` and go looking for our saved `ComponentName`. If that is not found — `getString()` returns `null` for our `PREF_TO_LAUNCH` — then the user has not chosen an activity from the Sidecar `MainActivity`. So, we show a `Toast` to let the user know that they need to choose an activity.

The `setGravity()` call in the `toast()` method shoves our `Toast` over into a lower corner of the screen, to try to get it a bit out of the way of the opened notification shade, as the shade will float over the `Toast`. This approach only works well on wider screens. A production-quality app would do something else here.

Given that we have the `String` representation of the `ComponentName`, we `unflattenFromString()` to get the `ComponentName` back, then put that `ComponentName` into an `Intent`. In most cases, we will then add the multiple-instance flags to the `Intent` (`FLAG_ACTIVITY_NEW_TASK`, `FLAG_ACTIVITY_MULTIPLE_TASK`, and `FLAG_ACTIVITY_LAUNCH_ADJACENT`), then start the activity.

Forcing Activities to Resize

However, if the activity in question has `android:resizeableActivity` set to `false`, we have more work to do.

First, to determine if that is the case for this activity, we have to access a hidden `resizeMode` field inside the `ActivityInfo` object. `ActivityInfo` contains a lot of the information from the `<activity>` element, though not all of it is part of the public Android SDK. Hacking into SDK objects using reflection **is strongly discouraged**,

as there is no guarantee that this field will exist on all devices, courtesy of firmware modifications. Using reflection this way is a great way to get a lot of customer service complaints from users about your app crashing. Regardless, we use the technique here. Based on the `ActivityInfo` source code, `android:resizeableActivity="false"` turns into a `resizeMode` of 0. So, the activity is resizable if `resizeMode` is anything other than 0.

However, as was discussed earlier in this chapter, `android:resizeableActivity` is only honored if your activity is the root of the task. For resizable activities, then, we can just launch the activity directly, but for non-resizable activities, we need to arrange to have *something else* be the task root. To that end, we have a `TaskRootActivity` that simply takes a supplied `Intent` (via an `EXTRA_INTENT` extra), starts an activity using that `Intent`, and finishes:

```
package com.commonsware.android.sidecar;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class TaskRootActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    startActivity((Intent)getIntent().getParcelableExtra(Intent.EXTRA_INTENT));
    finish();
  }
}
```

(from Introspection/Sidecar/app/src/main/java/com/commonsware/android/sidecar/TaskRootActivity.java)

`TaskRootActivity` is set up in the manifest to have a theme of `Theme.Translucent.NoTitleBar`, so the user will not usually see the activity, just the one that it starts. However, that is sufficient to allow `TaskRootActivity` to decide the resize rules for the task, overriding those of the user's chosen activity.

So, for non-resizable activities, `SidecarTileService` wraps the real `Intent` in one for `TaskRootActivity`, sets the multiple-instance flags, and starts that activity, which in turn starts the real activity in the separate pane.

Breaking the Sidecar

So, what can we learn from Sidecar about these other apps that do this sort of thing?

If your activity is exported — as are launcher activities — then there is little that you can do to stop other apps from launching your activity in a separate task to enable this sort of multi-window/multi-instance behavior. One thing that *does* stop it is to have `android:launchMode` of `singleInstance` or `singleTask`. That at least prevents multiple instances of your activity being launched in parallel, as it overrides the `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_MULTIPLE_TASK` flags.

Similarly — as we saw earlier in the chapter — there is nothing much that you can do to stop some other app from launching your activity in an existing task, thereby blocking you from controlling whether your activity gets resized.

Ideally, you do not worry much about either of these things, but instead set up your app to be able to work acceptably in these cases. Again, it is well within reason that device manufacturers will start offering this sort of "start another copy of the app" feature to their users. Sidecar, in this respect, can serve as a testbed for how well your app behaves when the user does this sort of thing.

Supporting Legacy Proprietary Multi-Window

As noted earlier in the chapter, some manufacturers experimented with multi-window implementations prior to Google rolling it into Android 7.0. These come in three flavors:

- Some, like Jide's Remix OS, require no developer work. Apps just show up in windows. There might be compatibility concerns, particularly since many Remix OS-powered devices have keyboards and mice. However, there is no up-front requirement to do something to opt into participating in the multi-window experience.
- Some, like Samsung's and LG's, require minor modification of your app's manifest, but no other specific work.
- Some, like SONY's "small apps", require a proprietary SDK, and therefore are somewhat more work. And, in the specific case of SONY's implementation, it has been discontinued.

Since the manifest-only modifications are "low-hanging fruit", giving your users some benefit with little additional work, let's take a look at how to enable those. The

MultiWindow/Legacy sample application is a clone of the `OptIn` sample, adjusted to allow pre-N versions of Android to run the app, and adjusted to support Samsung's and LG's legacy multi-window implementation.

The changes lie purely in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.multiwindow"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true" />

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:resizeableActivity="true"
    android:theme="@style/Theme.Apptheme">
    <uses-library
      android:name="com.sec.android.app.multiwindow"
      android:required="false" />

    <meta-data
      android:name="com.sec.android.support.multiwindow"
      android:value="true" />
    <meta-data
      android:name="com.lge.support.SPLIT_WINDOW"
      android:value="true" />

    <activity android:name="MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
        <category android:name="android.intent.category.MULTIWINDOW_LAUNCHER" />
      </intent-filter>
      <layout android:minWidth="480dp" />
    </activity>
  </application>

</manifest>
```

(from MultiWindow/Legacy/app/src/main/AndroidManifest.xml)

LG requires only one thing: a `<meta-data>` element in the `<application>`, with a name set to `com.lge.support.SPLIT_WINDOW` and a value of `true`. It assumes that your launcher activity (or activities) are suitable for showing in a split screen view.

Samsung requires three things:

- A similar `<meta-data>` element, setting `com.sec.android.support.multiwindow` to `true`
- A `<uses-library>` element, pulling in a firmware-supplied `com.sec.android.app.multiwindow` library, if it is available (`android:required="false"`)
- `<category android:name="android.intent.category.MULTIWINDOW_LAUNCHER" />` added to any `<intent-filter>` of any activity that you want to be able to be launched into the split-screen environment

In this case, we have only one activity, so it now has two `<category>` elements, for `LAUNCHER` and `MULTIWINDOW_LAUNCHER`. Note that while `android.intent.category.MULTIWINDOW_LAUNCHER` has the `android.intent.category` namespace, and not something like `com.sec.android`, `MULTIWINDOW_LAUNCHER` is not part of standard Android.

For supported devices from those manufacturers, your app will be available for split-screen use:

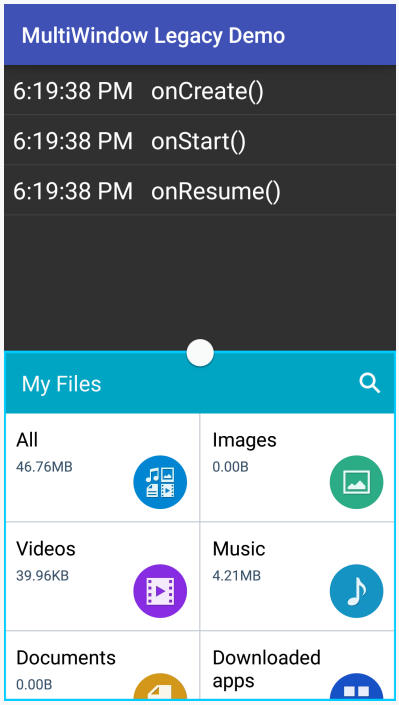


Figure 442: Samsung Legacy Split-Screen Mode

In the case of Samsung, it may also be available as a popup floating window:

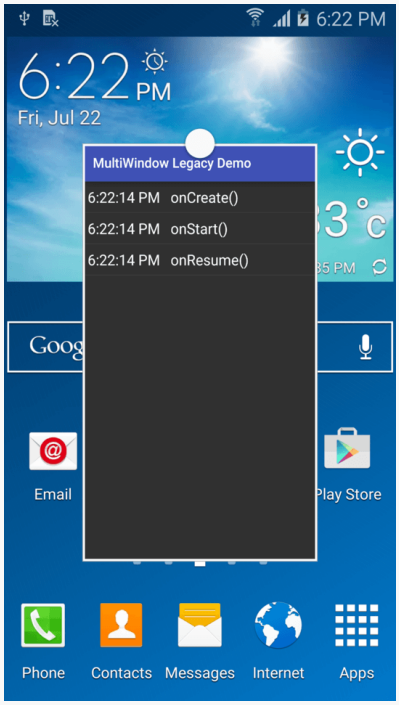


Figure 443: Samsung Legacy Popup Window Mode

One notable difference between these implementations and Android 7.0's multi-window implementation comes with lifecycle events. Android 7.0 will call lifecycle methods on your activity as appropriate during the transition to and from multi-window mode, and sometimes based on size changes of your window. Neither Samsung's nor LG's legacy multi-window does this.

Freeform Multi-Window Mode

Freeform multi-window mode — with desktop-style overlapping resizable windows — is not presently available in an official fashion. However, there are tricks for making it work, even without rooting a device. As a result, some power users will start playing with your app in freeform windows, despite the lack of official support.

Playing with Freeform

If you would like to play with the unofficial freeform multi-window mode, there are a few ways of going about it.

The Taskbar App

There are some apps out on the Play Store and other distribution channels that allow you to launch apps in freeform mode, such as Braden Farmer's Taskbar app.

Freeform support requires you to go into the "Developer options" area of the Settings app and enable "Force activities to be resizable":

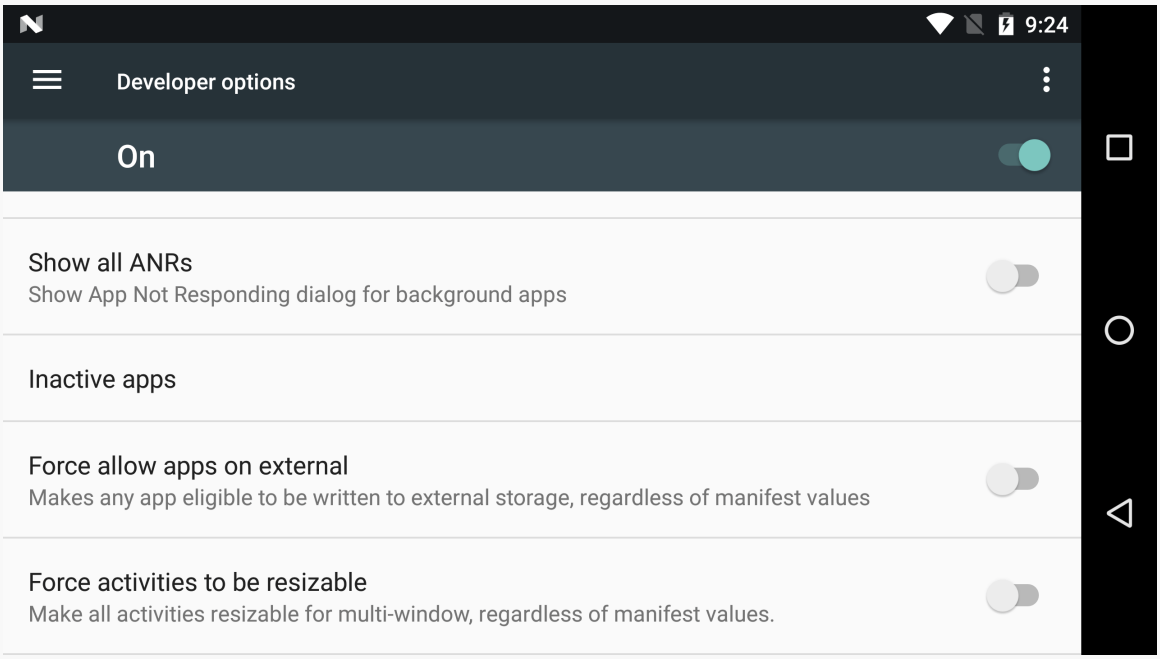


Figure 444: Resizable Windows Developer Setting

If you allow Taskbar to be your home screen, it can launch freeform windows reliably. If you use Taskbar solely as a floating launcher bar – the default — it can launch freeform windows from the overview screen (a.k.a., recent-tasks list) or the home screen, but not elsewhere, and occasionally there will be hiccups where the activity will be launched normally (not freeform).

The adb Setting

If you prefer, you can enable freeform multi-window mode for your device or emulator via:

```
adb shell settings put global enable_freeform_support 1
```

Then, after a reboot, when you visit the overview screen, you will see a window-inset icon in the title bar of the overview cards:

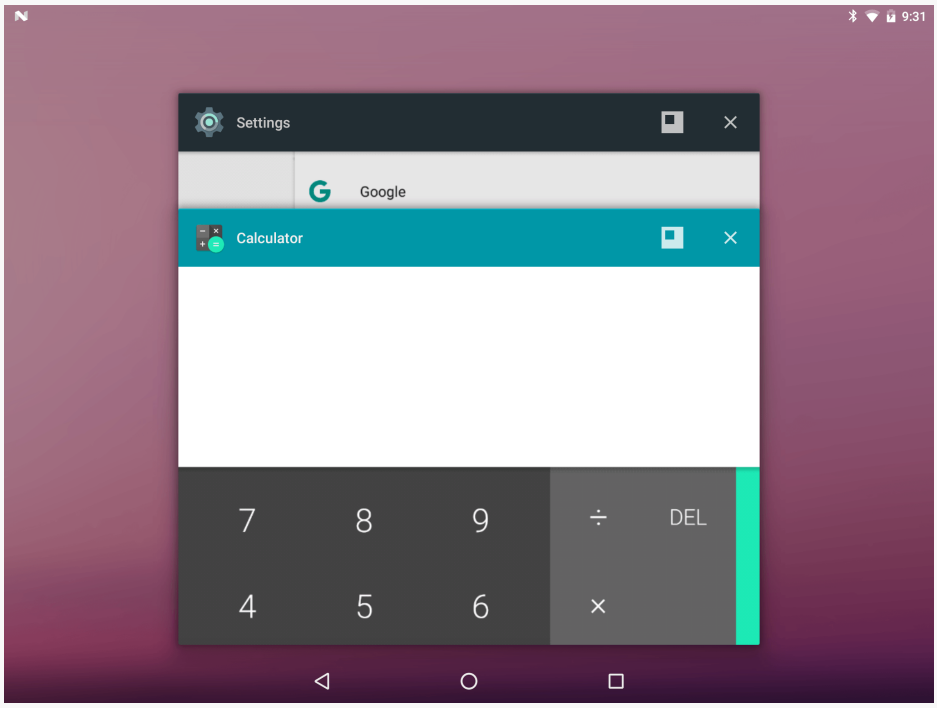


Figure 445: Overview Screen, Showing Freeform Title Bar Icons

Tapping that freeform-window icon will open that particular task in a freeform window:

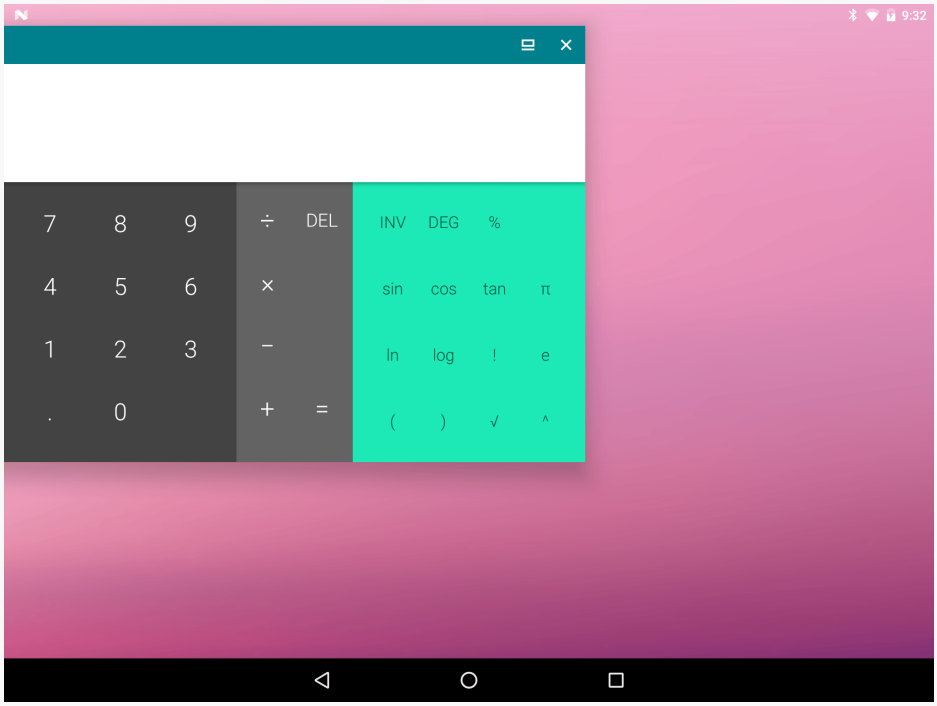


Figure 446: Calculator in Freeform Window

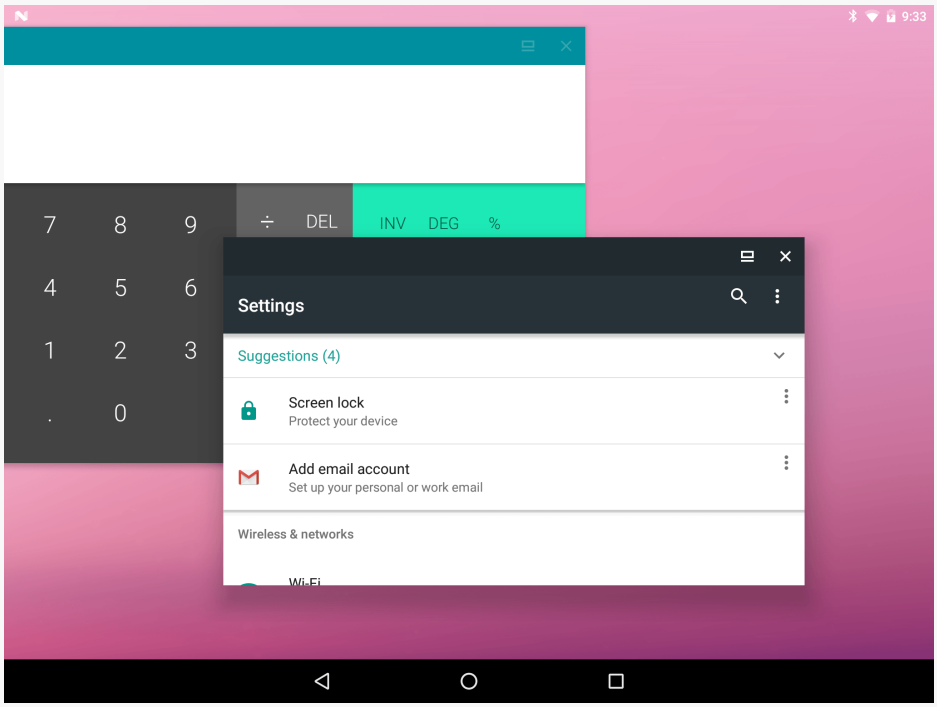


Figure 447: Calculator and Settings in Freeform Window

To undo this change, run:

```
adb shell settings put global enable_freeform_support 0
```

After a reboot, platform-level freeform multi-window support will be disabled again.

The Freecar App

Braden Farmer's Taskbar app is open source. All that it is doing is using documented APIs to describe how a window should be displayed in freeform mode, so one can imagine that home screen app developers might start offering a similar capability.

The `Introspection/Freecar` sample application is a simpler demonstration of launching freeform windows on devices where either "Force activities to be resizable" or the `enable_freeform_support` global setting is enabled. This is a clone of the earlier `Sidecar` sample, except this time the notification shade tile will launch the chosen activity in freeform mode.

API Level 16 added a version of `startActivity()` that takes both an `Intent` and a `Bundle` as parameters. The `Bundle` is a way of providing additional information to Android describing how the activity should be started. That `Bundle` is typically created via an `ActivityOptions` object, where you configure a set of options, call `toBundle()` on it to convert the options into a `Bundle`, then pass that `Bundle` to `startActivity()`.

On API Level 24+ devices, `setLaunchBounds()` on `ActivityOptions` indicates that you would like the window to be launched in freeform mode at a particular location, described via a `Rect`.

`FreecarTileService` — the `TileService` for our notification shade tile — responds to a click by getting the chosen activity details and using `setLaunchBounds()` as part of `startActivity()`:

```
@Override
public void onClick() {
  super.onClick();

  SharedPreferences prefs=
    PreferenceManager.getDefaultSharedPreferences(this);
  String cnFlat=prefs.getString(PREF_TO_LAUNCH, null);

  if (cnFlat!=null) {
```

```
    ComponentName cn=ComponentName.unflattenFromString(cnFlat);

    try {
      ActivityInfo info=getPackageManager().getActivityInfo(cn, 0);
      ActivityInfo.WindowLayout layout=info.windowLayout;
      Intent i=
        new Intent()
          .setComponent(cn)
          .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
      Point size=new Point();

      getSystemService(DisplayManager.class)
        .getDisplay(Display.DEFAULT_DISPLAY)
        .getSize(size);

      if (layout==null) {
        size.x=size.x/2;
        size.y=size.y/2;
      }
      else {
        if (layout.widthFraction>0.0f) {
          size.x=
            Math.max(layout.minWidth,
              (int)(size.x*layout.widthFraction));
        }
        else {
          size.x=layout.width;
        }

        if (layout.heightFraction>0.0f) {
          size.y=
            Math.max(layout.minHeight,
              (int)(size.y*layout.heightFraction));
        }
        else {
          size.y=layout.height;
        }
      }

      ActivityOptions opts=
        ActivityOptions
          .makeBasic()
          .setLaunchBounds(new Rect(0, 0, size.x, size.y));

      startActivity(i, opts.toBundle());
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
```

```
          "Exception trying to launch activity", e);

        toast(R.string.msg_sorry);
      }
    }
    else {
      toast(R.string.msg_choose);
    }
  }
```

(from Introspection/Freecar/app/src/main/java/com/commonsware/android/freecar/FreecarTileService.java)

There are a few possibilities for the values for the window size:

- If the activity does not have a `<layout>` element as a child of its `<activity>` element in the manifest, we have no idea how big the activity should be, so we set it to be half of the screen size.
- If the activity has a `<layout>` element, use its `android:width`, `android:height`, `android:minWidth`, and `android:minHeight` values, along with the window size, to calculate the desired size.

We find out about those values by getting the `ActivityInfo` for this activity from the `PackageManager` and looking at the `WindowLayout` provided via the `windowLayout` field. This will be `null` if we have no `<layout>` element; otherwise, it will contain the values specified in that `<layout>` element.

Freeform and Your App

While basic freeform multi-window support shipped with Android 7.0, it is unclear how close it is to something that device manufacturers might enable. It is also unclear whether Google endorses this or whether the 7.0 freeform multi-window support is merely for experimentation purposes.

As such, probably it is not worthwhile to spend a lot of time testing your app in freeform multi-window mode. Once we have a better idea of whether this is something that substantial numbers of users will have access to, we will be able to better judge how much testing time is warranted.

Picture-in-Picture

Android 7.0 added picture-in-picture support for apps, but only on Android TV. Android 8.0 extended this to mobile devices. An activity can elect to support picture-in-picture and can elect to move itself into picture-in-picture mode.

While in picture-in-picture mode, the activity is considered to be paused, as it no longer receives input events. However, you can add "actions", with toolbar-style icons and associated `PendingIntents`, that will get triggered when the user taps on the window to show the toolbar, then taps on an action:

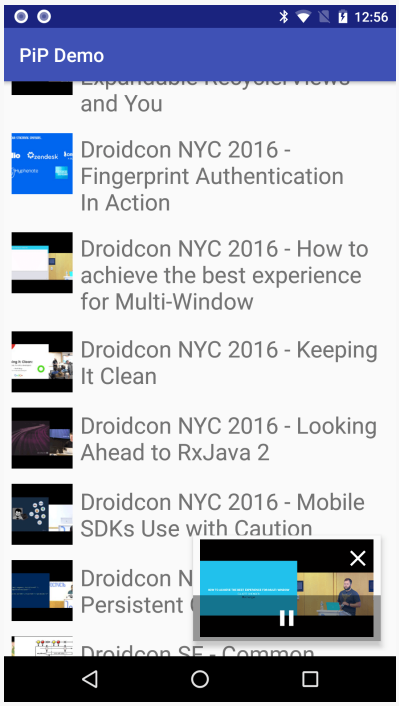


Figure 448: Picture-in-Picture, Showing Single-Item Toolbar and X

If the user exits the picture-in-picture activity via the X icon, the activity is not destroyed. Rather, it returns to normal size. However, if you follow the documented instructions, and use `android:launchMode="singleTask"` on this activity, the activity's task will be moved to the background when the user exits it. The result is that the activity will be stopped (as it is no longer visible), but the activity remains, and the user could return to it through the overview screen if desired. Whether this is a good idea or not remains to be seen.

The MultiWindow/PiP sample project illustrates how this works. This sample app is a clone of a sample from the chapter on RecyclerView, where we use the MediaStore to pull up a list of videos, then view the video when the user taps on the list row. In the original example, we play the video using some default video player (e.g., ACTION_VIEW Intent on the video Uri). In this revised example, we have our own crude video player via a VideoView in a custom VideoPlayerActivity. That activity supports picture-in-picture mode.

To opt into picture-in-picture support, you first need to have android:supportsPictureInPicture="true" on the <activity> element in the manifest:

```
    <activity
      android:name=".VideoPlayerActivity"
      android:configChanges="screenSize|smallestScreenSize|screenLayout|orientation"
      android:launchMode="singleTask"
      android:supportsPictureInPicture="true"
      android:theme="@style/Theme.Apptheme.NoActionBar" />
```

(from MultiWindow/PiP/app/src/main/AndroidManifest.xml)

The documentation also recommends that you use android:launchMode="singleTask", so that there is at most one instance of this activity outstanding at a time. A practical requirement is that this activity cannot support the action bar, as you cannot get rid of it when you move into picture-in-picture mode. And, particularly since this activity will be handling video playback, opting into handling some of the configuration changes tied to screen rotations may be appropriate.

Simply having android:supportsPictureInPicture="true" means that the activity *supports* picture-in-picture mode. However, by default, it is not *in* picture-in-picture mode. To move into that mode, upon user request, call enterPictureInPictureMode(). In the sample app, the VideoPlayerActivity has a floating action button (FAB) that, when clicked, moves the activity into picture-in-picture mode:

```
@Override
public void onClick(View view) {
  aspectRatio=new Rational(video.getWidth(), video.getHeight());
  enterPictureInPictureMode(updateActions());
}
```

(from MultiWindow/PiP/app/src/main/java/com/commonsware/android/recyclerview/videolist/VideoPlayerActivity.java)

We want to set up the aspect ratio of the picture-in-picture window to match that of the video. By this time, our `VideoView` has the proper dimensions, so we can calculate that aspect ratio. However, rather than using a `float` or `double`, the picture-in-picture system uses a `Rational`. This class, added to API Level 21, captures the numerator and denominator of a rational number. Among other things, it postpones or eliminates the need to switch to the imprecision of floating-point numbers.

The `enterPictureInPictureMode()` method takes a `PictureInPictureParams` object, which we build here via the `updateActions()` method:

```
private PictureInPictureParams updateActions() {
  ArrayList<RemoteAction> actions=new ArrayList<>();

  if (video.isPlaying()) {
    actions.add(buildRemoteAction(REQUEST_PAUSE,
      R.drawable.ic_pause_white_24dp, R.string.pause, R.string.pause_desc));
  }
  else {
    actions.add(buildRemoteAction(REQUEST_PLAY,
      R.drawable.ic_play_arrow_white_24dp, R.string.play, R.string.play_desc));
  }

  return(new PictureInPictureParams.Builder()
    .setAspectRatio(aspectRatio)
    .setActions(actions)
    .build());
}
```

(from MultiWindow/PiP/app/src/main/java/com/commonsware/android/recyclerview/videolist/VideoPlayerActivity.java)

Partly, the `PictureInPictureParams` object captures and uses that `aspectRatio` `Rational` value. Mostly, though, we use `PictureInPictureParams` for a roster of `RemoteAction` objects, representing the toolbar-style buttons. Here, depending upon whether or not the video is currently playing, we set an action for a play or a pause operation. `buildRemoteAction()` wraps our desired icon and other information in a `RemoteAction`:

```
private RemoteAction buildRemoteAction(int requestCode, int iconId,
                                       int titleId, int descId) {
  Intent i=new Intent(this, RemoteActionReceiver.class)
    .putExtra(EXTRA_REQUEST, requestCode);
  PendingIntent pi=PendingIntent.getBroadcast(this, requestCode, i, 0);
  Icon icon=Icon.createWithResource(this, iconId);
  String title=getString(titleId);
  String desc=getString(descId);

  return(new RemoteAction(icon, title, desc, pi));
}
```

(from MultiWindow/PiP/app/src/main/java/com/commonsware/android/recyclerview/videolist/VideoPlayerActivity.java)

Of note, we have the actions trigger a `RemoteActionReceiver` via a broadcast `PendingIntent`. We could use a dynamically-registered `BroadcastReceiver`, but those are always exported, and that makes securing them a bit tricky. Instead, `RemoteActionReceiver` is registered (but not exported) in the manifest. All it does is forward the received `Intent` to the rest of the app via greenrobot's EventBus:

```
package com.commonsware.android.recyclerview.videolist;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import org.greenrobot.eventbus.EventBus;

public class RemoteActionReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    EventBus.getDefault().post(intent);
  }
}
```

(from MultiWindow/PiP/app/src/main/java/com/commonsware/android/recyclerview/videolist/RemoteActionReceiver.java)

`VideoPlayerActivity` picks up that event, looks at the `EXTRA_REQUEST` extra, pauses or resumes the video based on the request, and updates the actions to reflect the new video state:

```
  @Subscribe(threadMode =ThreadMode.MAIN)
  public void onReceive(Intent intent) {
    int requestCode=intent.getIntExtra(EXTRA_REQUEST, -1);

    if (requestCode==REQUEST_PAUSE) {
      video.pause();
    }
    else if (requestCode==REQUEST_PLAY) {
      video.start();
    }

    setPictureInPictureParams(updateActions());
  }
```

(from MultiWindow/PiP/app/src/main/java/com/commonsware/android/recyclerview/videolist/VideoPlayerActivity.java)

You can find out when an activity enters or leaves picture-in-picture mode by overriding `onPictureInPictureModeChanged()`. In `VideoPlayerActivity`, we use

this to hide and show the FAB, as the FAB is not needed when we are in picture-in-picture mode:

```
@Override
public void onPictureInPictureModeChanged(boolean isInPictureInPictureMode,
                                          Configuration cfg) {
  super.onPictureInPictureModeChanged(isInPictureInPictureMode, cfg);

  fab.setVisibility(isInPictureInPictureMode ? View.GONE : View.VISIBLE);
}
```

(from MultiWindow/PiP/app/src/main/java/com/commonsware/android/recyclerview/videolist/VideoPlayerActivity.java)

Note that while there is an `enterPictureInPicture()` mode, there is no equivalent way for you to programmatically leave picture-in-picture mode.

Multi-Display Support

Android 8.0 also gave us a new way to put content on an external display, such as a TV or projector attached to a mobile device. Formerly, we had to use `Presentation` and related classes. With Android 8.0+, we can use multi-window techniques to send an activity to the external display, which is far simpler. It involves three main steps:

1. Finding the `Display` that you want to send the other activity to,
2. Building an `ActivityOptions` that includes the display ID of that `Display`, and
3. Calling a version of `startActivity()` that accepts a `Bundle` created by `ActivityOptions` as a parameter, and using an `Intent` with the appropriate flags

Since an external display does not support any form of input, the net result is akin to picture-in-picture: we can show an activity, but the user cannot interact with it directly. In fact, the user has less ability to interact with it than the user does with an activity started in picture-in-picture mode. Primarily, this feature is designed for control being managed by a device's native form of input (e.g., touchscreen on typical phones and tablets), and with the external display being used purely for output (e.g., a movie, a presentation, a set of charts).

The `Presentation/MultiDisplay` sample project shows the basic technique. This is a clone of the `Presentation/Slides` sample project, which used a `ViewPager` and a `Presentation` to allow a presenter to show slides from a phone or tablet connected

to a projector. `Presentation/MultiDisplay` changes that to use a separate activity on Android 8.0+, routed to the projector, that displays the current slide.

Our two activities — `MainActivity` with the `ViewPager` and a `PresentationActivity` to show on the projector — are both in the manifest, with `PresentationActivity` set up to be locked to landscape orientation:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.preso.slides"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity
      android:name=".PresentationActivity"
      android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSize"
      android:label="@string/app_name"
      android:screenOrientation="landscape"
      android:theme="@style/PresoAppTheme" />
  </application>

</manifest>
```

(from Presentation/MultiDisplay/app/src/main/AndroidManifest.xml)

`MainActivity`, in `onCreate()`, sets up a `ViewPager` with a `SampleAdapter` that will show one slide per page. It also sets up a `TabLayout` from the CWAC-CrossPort library to use for the tabs. Most importantly, though, is that on Android 8.0+, it starts the work to see if we have a presentation display to use:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new SlidesAdapter(this);
    pager.setAdapter(adapter);
```

```
    TabLayout tabs=(TabLayout)findViewById(R.id.tabs);

    tabs.setupWithViewPager(pager);
    tabs.setTabMode(TabLayout.MODE_SCROLLABLE);
    tabs.addOnTabSelectedListener(this);

    if (iCanHazO()) {
      dm=getSystemService(DisplayManager.class);
      dm.registerDisplayListener(this, null);
      checkForPresentationDisplays();
    }
  }
```

(from Presentation/MultiDisplay/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

`iCanHazO()` relies on `Build.VERSION.CODENAME` to determine if we are on Android 8.0+, given that developer previews use the prior API level for `Build.VERSION.SDK_INT`:

```
  public static boolean iCanHazO() {
    return(Build.VERSION.SDK_INT>=Build.VERSION_CODES.O);
  }
```

(from Presentation/MultiDisplay/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

If we are on Android 8.0+, `onCreate()` obtains a `DisplayManager`, registers the activity itself as a listener for changes in the state of available displays, and calls `checkForPresentationDisplays()`. That method, in turn, will see if we have a `DISPLAY_CATEGORY_PRESENTATION` display:

```
  private void checkForPresentationDisplays() {
    if (dm!=null && presoItem!=null) {
      Display[] displays=
        dm.getDisplays(DisplayManager.DISPLAY_CATEGORY_PRESENTATION);

      if (displays.length>0) {
        presoItem.setEnabled(true);
        presoDisplay=displays[0];
      }
      else {
        presoItem.setEnabled(false);
        presoDisplay=null;
      }
    }
  }
```

(from Presentation/MultiDisplay/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

If we do, we hold onto the `Display` object identifying that display, plus enable an action bar item. When that action bar item is clicked, we want to start `PresentationActivity` on that `Display`:

```
      Intent i=
        new Intent(this, PresentationActivity.class)
          .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_MULTIPLE_TASK);
      Bundle opts=ActivityOptions
        .makeBasic()
        .setLaunchDisplayId(presoDisplay.getDisplayId())
        .toBundle();

      startActivity(i, opts);
```

(from Presentation/MultiDisplay/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

For starting one of our own activities into the external display, we need to add the `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_MULTIPLE_TASK` flags to our `Intent`, akin to what we would use with `FLAG_ACTIVITY_LAUNCH_ADJACENT` to launch our activity into a separate window. For starting a third-party app's activity into the external display, we might be able to skip those flags, depending on the task settings in the manifest of the activity that we are starting.

`ActivityOptions` is a class for configuring how an activity should be started. Mostly, it is used for controlling transition effects. However, Android 8.0 extended it with `setLaunchDisplayId()`, which takes the display ID of the `Display` on which we want to start this activity. In our case, that is the presentation `Display` that we obtained from the `DisplayManager`. `ActivityOptions` can then be converted to a `Bundle`, which gets passed to `startActivity()` along with our `Intent`.

The net result: the user can only use this action bar item on Android 8.0+ devices connected to a presentation display, and on those devices, `PresentationActivity` is shown on that display. This can be tested on O Developer Preview devices using the simulated secondary display option in Developer Options in Settings.

However, we still need a way for the user to control the contents of the `PresentationActivity`. The idea here is the same as in the original `Presentation/Slides` sample: the `PresentationActivity` should show the currently-selected slide in the `ViewPager`.

To that end, `onCreate()` of `MainActivity` registered itself to respond to `TabLayout` events. In `onTabSelected()`, we post a sticky message to greenrobot's EventBus, containing the drawable resource ID of the currently-shown slide:

```
  @Override
  public void onTabSelected(TabLayout.Tab tab) {
    int position=tab.getPosition();
    int resourceId=adapter.getPageResource(position);

    EventBus
      .getDefault()
      .postSticky(new SlideChangedEvent(resourceId));
  }
```

(from Presentation/MultiDisplay/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

`PresentationActivity`, in turn, registers for that sticky event and shows that slide, or the first slide if there is no such event:

```
package com.commonsware.android.preso.slides;

import android.app.Activity;
import android.os.Bundle;
import android.support.annotation.Nullable;
import android.widget.ImageView;
import org.greenrobot.eventbus.EventBus;
import org.greenrobot.eventbus.Subscribe;
import org.greenrobot.eventbus.ThreadMode;

public class PresentationActivity extends Activity {
  private ImageView slide;

  @Override
  protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_preso);

    slide=(ImageView)findViewById(R.id.slide);

    MainActivity.SlideChangedEvent event=
      EventBus.getDefault().getStickyEvent(MainActivity.SlideChangedEvent.class);

    if (event==null) {
      slide.setImageResource(R.drawable.img0);
    }
  }

  @Override
  protected void onStart() {
    super.onStart();

    EventBus.getDefault().register(this);
  }

  @Override
  protected void onStop() {
    EventBus.getDefault().unregister(this);

    super.onStop();
  }
```

```
  @Subscribe(sticky = true, threadMode =ThreadMode.MAIN)
  public void onSlideChanged(MainActivity.SlideChangedEvent event) {
    slide.setImageResource(event.resourceId);
  }
}
```

(from Presentation/MultiDisplay/app/src/main/java/com/commonsware/android/preso/slides/PresentationActivity.java)

Using a sticky event means that `MainActivity` does not really care if `PresentationActivity` is currently available or not. It just fires off the events, knowing that whenever `PresentationActivity` is started, it can pick up the latest such event, plus any new ones raised while `PresentationActivity` is around.

Not only can use multi-display mode as a replacement for `Presentation`, but you can do something that was never officially supported with `Presentation`: control the external display while allowing some other activity to take over the primary display. This is illustrated in the `Presentation/MultiDisplayDetached` sample app. This is a clone of the `MultiDisplay` one, but set up where `MainActivity` goes away after kicking off the presentation.

`MainActivity` has the same action bar items as before, but this time, when the user starts the presentation, we handle that event slightly differently:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.present:
        Intent i=
          new Intent(this, PresentationActivity.class)
            .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_MULTIPLE_TASK);
        Bundle opts=ActivityOptions
          .makeBasic()
          .setLaunchDisplayId(presoDisplay.getDisplayId())
          .toBundle();

        startActivity(i, opts);
        showNotification();
        finish();

        break;

      case R.id.first:
        pager.setCurrentItem(0);
        break;

      case R.id.last:
        pager.setCurrentItem(adapter.getCount() - 1);
        break;
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from Presentation/MultiDisplayDetached/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

After starting the `PresentationActivity` in multi-display mode, we `finish()` the `MainActivity`. However, we also call `showNotification()`, which brings up a `Notification` to use to control the slides:

```
  private void showNotification() {
    NotificationCompat.Builder b=
      new NotificationCompat.Builder(this)
        .setOngoing(true)
        .setContentTitle("Presentation!")
        .setSmallIcon(android.R.drawable.stat_notify_more)
        .addAction(android.R.drawable.ic_media_previous,
          getString(R.string.action_previous), buildPreviousPendingIntent())
        .addAction(android.R.drawable.ic_media_next,
          getString(R.string.action_next), buildNextPendingIntent())
        .addAction(android.R.drawable.ic_media_pause,
          getString(R.string.action_stop), buildStopPendingIntent());

    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    mgr.notify(1337, b.build());
  }

  private PendingIntent buildPreviousPendingIntent() {
    Intent i=new Intent(this, ControlReceiver.class).putExtra(EXTRA_DELTA, -1);

    return(PendingIntent.getBroadcast(this, PI_PREVIOUS, i, 0));
  }

  private PendingIntent buildNextPendingIntent() {
    Intent i=new Intent(this, ControlReceiver.class).putExtra(EXTRA_DELTA, 1);

    return(PendingIntent.getBroadcast(this, PI_NEXT, i, 0));
  }

  private PendingIntent buildStopPendingIntent() {
    Intent i=new Intent(this, ControlReceiver.class).putExtra(EXTRA_STOP, true);

    return(PendingIntent.getBroadcast(this, PI_STOP, i, 0));
  }
```

(from Presentation/MultiDisplayDetached/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

Those broadcasts go to a `ControlReceiver`, which is registered in the manifest and simply forwards those broadcasts on an event bus:

```
package com.commonsware.android.preso.slides;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import org.greenrobot.eventbus.EventBus;
```

```
public class ControlReceiver extends BroadcastReceiver {
  static final String EXTRA_DELTA="delta";
  static final String EXTRA_STOP="orMyMomWillShoot";

  @Override
  public void onReceive(Context context, Intent intent) {
    EventBus.getDefault().post(intent);
  }
}
```

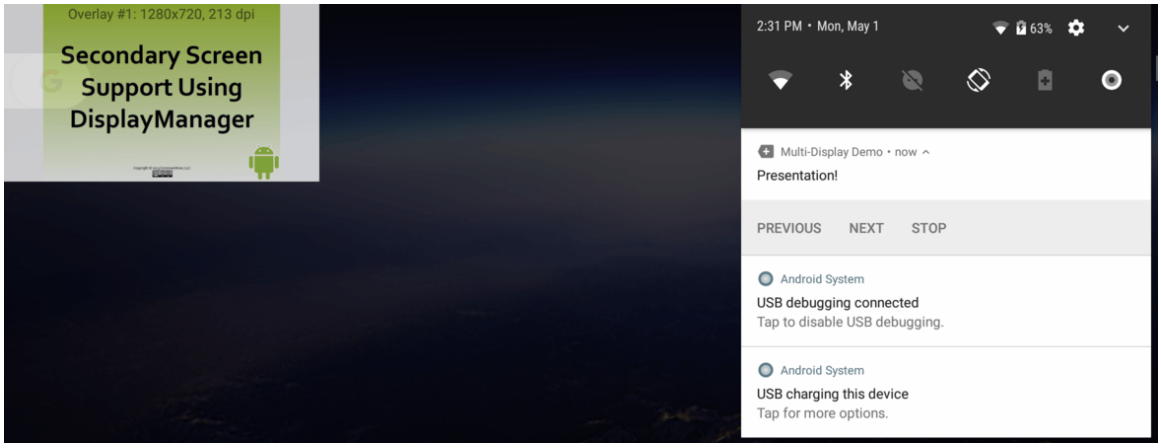
(from Presentation/MultiDisplayDetached/app/src/main/java/com/commonsware/android/preso/slides/ControlReceiver.java)

`PresentationActivity` can then listen for that event, along with the `SlidePositionEvent`, using each to update the visible slide:

```
  @Subscribe(sticky=true, threadMode=ThreadMode.MAIN)
  public void onSlideChanged(SlidePositionEvent event) {
    position=event.position;
    updateSlide();
  }

  @Subscribe(threadMode=ThreadMode.MAIN)
  public void onBroadcast(Intent i) {
    if (i.getBooleanExtra(EXTRA_STOP, false)) {
      ((NotificationManager)getSystemService(NOTIFICATION_SERVICE)).cancelAll();
      finish();
    }
    else {
      int delta=i.getIntExtra(EXTRA_DELTA, 0);

      if (position+delta>=0 && position+delta<SlidesAdapter.SLIDES.length) {
        position+=delta;
        updateSlide();
      }
    }
  }

  private void updateSlide() {
    slide.setImageResource(SlidesAdapter.SLIDES[position]);
  }
```

(from Presentation/MultiDisplayDetached/app/src/main/java/com/commonsware/android/preso/slides/PresentationActivity.java)

The result is that the slides remain visible — and controllable via the `Notification` — even though `MainActivity` is gone:


Overlay #1: 1280x720, 213 dpi
Secondary Screen Support Using DisplayManager
2:31 PM • Mon, May 1
63%
Multi-Display Demo • now
Presentation!
PREVIOUS
NEXT
STOP
Android System
USB debugging connected
Tap to disable USB debugging.
Android System
USB charging this device
Tap for more options.

Advanced ConstraintLayout

The original chapter on `ConstraintLayout` covered some basic uses of this container class, particularly how it can be used in the place of classic containers like `LinearLayout` and `RelativeLayout`. In this chapter, we will explore other features that `ConstraintLayout` offers and other use cases for this container.

This chapter examines some more layouts from the `Containers/Sampler` sample project.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the original chapter on `ConstraintLayout`.

Disclosing Your Bias

For a given axis, there are roughly three models for how you use constraints for a given widget:

- Anchor one side of the widget to something based upon that axis (e.g., `app:layout_constraintStart_toStartOf="parent"`)
- Anchor both sides of the widget to something based upon that axis, with the size on that axis (e.g., `android:layout_width` for the horizontal axis) set to `0dp`
- Anchor both sides of the widget to something based upon that axis, with the size on that axis set to `wrap_content` or some non-zero dimension

In the first case, the widget is simply attached to the anchor point, taking any margins into account. In the second case, the widget is stretched between the two anchor points.

The default for the third case is for the widget to be centered within the available space between the two anchor points, assuming that its size on that axis is smaller than the available space. However, we can use "bias" to slide the widget along that axis away from the center point, so it appears closer to one end than the other.

Centering a widget within a `RelativeLayout` is a matter of using `android:layout_centerInParent="true"`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="@string/button"/>

</RelativeLayout>
```

(from Containers/Sampler/app/src/main/res/layout/center_rl.xml)

Centering a widget within a `ConstraintLayout` is more verbose, requiring you to anchor all four sides of the widget to the four edges of the `ConstraintLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

```
</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/center_cl.xml)

However, that verbosity also allows us to apply biases, via `app:layout_constraintHorizontal_bias` and `app:layout_constraintVertical_bias`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:tools="http://schemas.android.com/tools"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.33"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.32999998"
    tools:layout_editor_absoluteX="98dp" />

</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/bias_cl.xml)

The default values for the biases are 0.5, meaning that the center point of the widget is centered evenly between the two anchor points. A bias of less than 0.5 slides the widget towards the beginning of that axis (towards the start size of the horizontal axis or the top of the vertical axis). A bias of higher than 0.5 slides the widget towards the end of that axis.

So, for bias values of 0.33 along each axis, the `Button` is slid towards the start and top:

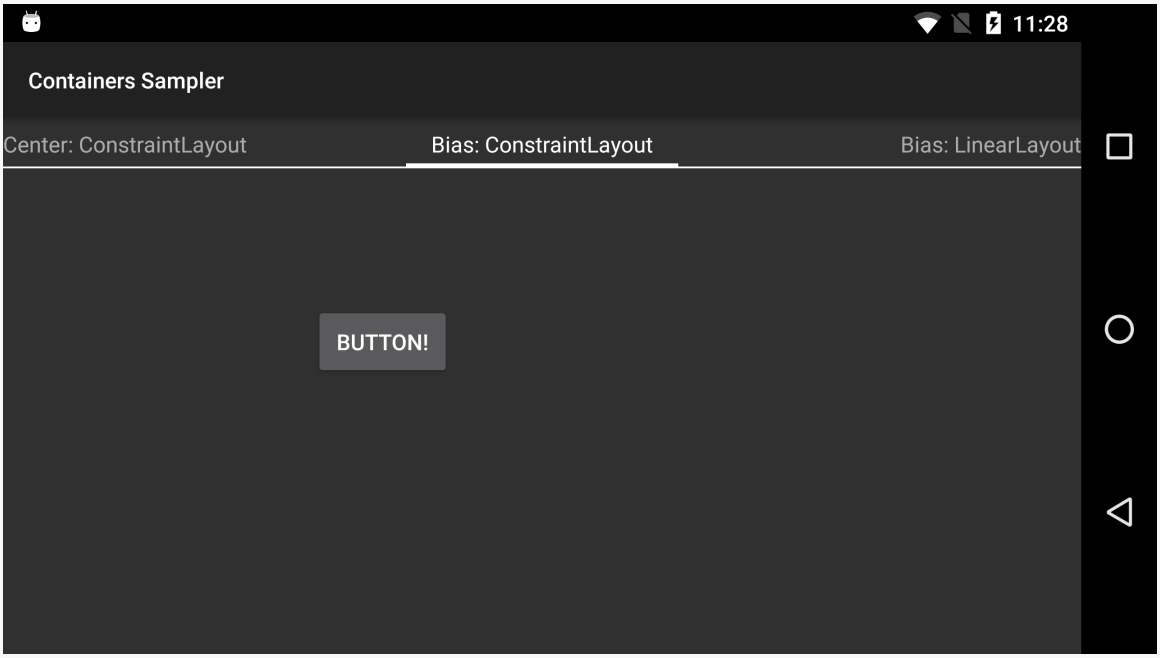


Figure 449: Bias Sample, Using ConstraintLayout

It is impractical to replicate this using the classic container classes. `LinearLayout` comes closest, using something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:weightSum="100">

  <View
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="33" />

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="67"
    android:orientation="horizontal"
    android:weightSum="100">

    <View
      android:layout_width="0dp"
```

```
      android:layout_height="match_parent"
      android:layout_weight="33" />

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/button" />
  </LinearLayout>

</LinearLayout>
```

(from Containers/Sampler/app/src/main/res/layout/bias_ll.xml)

Here, we use empty `View` widgets as spacers, along with weights to get the proportions the way that we want. However, this puts a *corner* of the `Button` at the appropriate location, not its center. So, compared to the `ConstraintLayout` approach, the `LinearLayout` implementation is a bit closer to the center:

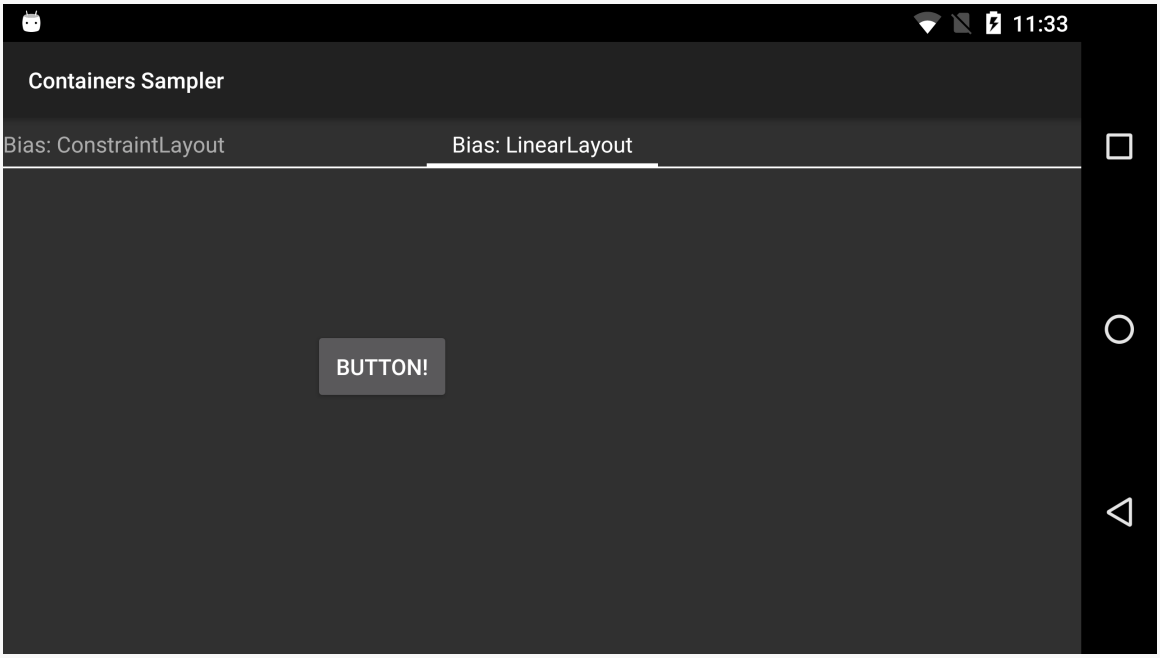


Figure 450: Bias Sample, Using LinearLayout

`ConstraintLayout` biases are not limited to constraints placed against the `ConstraintLayout` edges. You can apply biases based on constraints tied to guidelines or other widgets as well:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="@+id/guideline"
    app:layout_constraintHorizontal_bias="0.1"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="@+id/guideline"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.25" />

  <android.support.constraint.Guideline
    android:id="@+id/guideline"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintGuide_percent=".50" />

  <Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/btn_other"
    app:layout_constraintHorizontal_bias="0.8"
    app:layout_constraintLeft_toRightOf="@+id/button3"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="@+id/button3" />

</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/bias_peers_cl.xml)

Here we have a `ConstraintLayout` with two buttons. The “Button!” button has two biases:

- A vertical bias to place it at the 25% mark from the top
- A horizontal bias to place it at the 10% mark between the screen edge and a center `Guideline`

The “Other Button!” button has a horizontal bias to place it at the 80% mark between the edge of the first button and the screen edge.

The buttons are positioned with those biases applied:

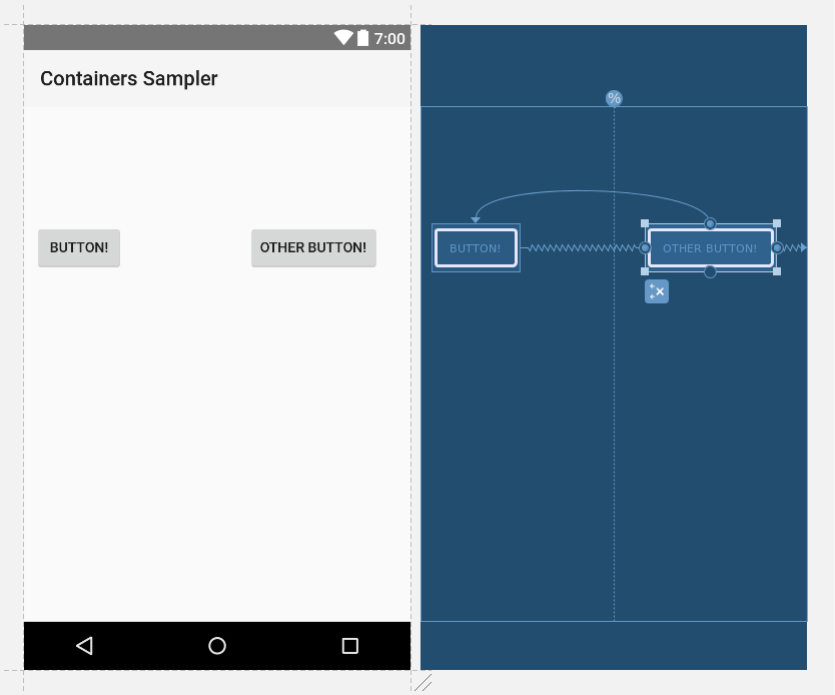


Figure 451: Biases with Widgets, Guidelines, and Edges

The configuration tool atop the Attributes pane lets you control the bias, by adjusting the slider for the bias to apply to that particular axis:

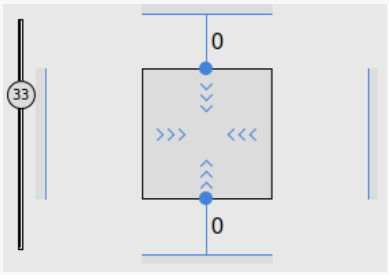


Figure 452: Configuration Thingy, Showing 33% Bias

Centering Yourself

Sometimes, we want to have one widget be centered upon another widget. This could be in either direction: horizontal centering or vertical centering.

The classic way to approximate this would involve you wrapping those two widgets in a container, then use `android:layout_gravity` to ask each of those widgets to center itself within the container:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="16dp"
    android:text="@string/button" />

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="16dp"
    android:text="@string/another_button" />

</LinearLayout>
```

(from Containers/Sampler/app/src/main/res/layout/center_align_ll.xml)

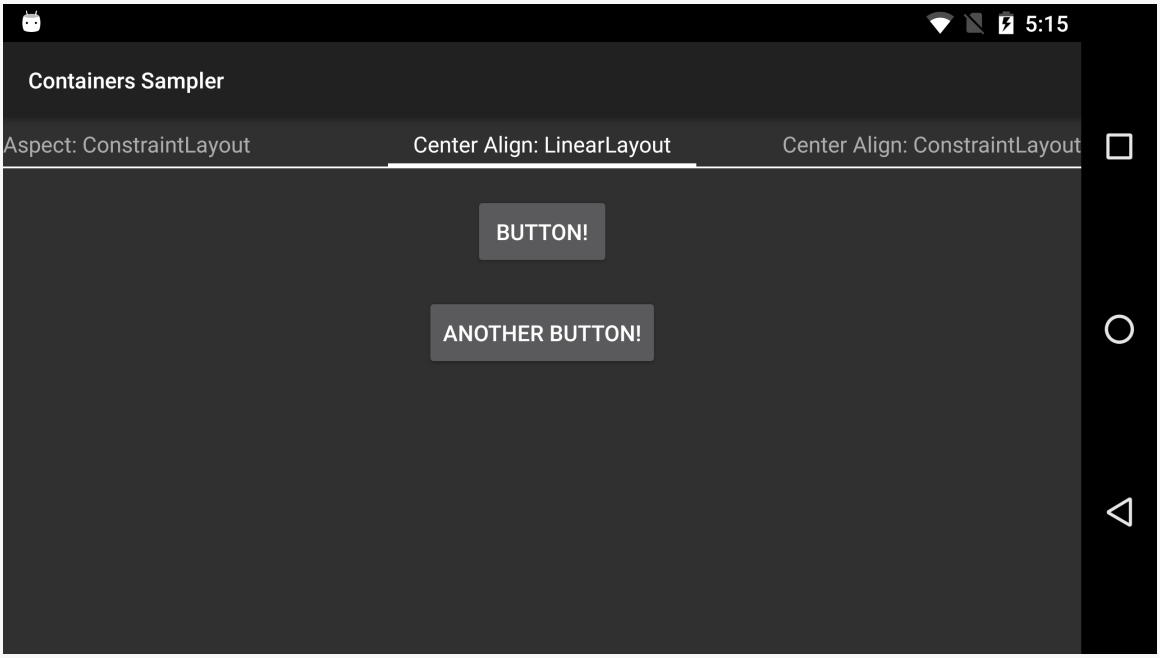


Figure 453: Center Alignment, Using LinearLayout

However, this approach has flaws:

- This actually centers the smaller widget on the larger one. In this case, the bottom `Button` has a larger caption, and so the smaller (top) `Button` is centered within the larger (bottom) `Button`.
- It requires that the two widgets be wrapped in a container, which may make it more difficult for those widgets to be positioned overall within the layout. They have to be moved in unison, rather than treated separately (other than for the center alignment aspect). This extra container also consumes heap size, rendering time, etc.

Using `ConstraintLayout`, we can just have rules that enforce the center alignment, along with all the rest of our rules, without the need for the additional container.

The trick is to simply have the dependent widget constrain its edges to match the edges of the independent widget. So, if we wanted to say that the bottom `Button` should be centered on the top `Button`, we get this:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
  android:layout_width="wrap_content"
  android:layout_height="wrap_content">

  <Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="@string/button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="@string/another_button"
    app:layout_constraintEnd_toEndOf="@id/button"
    app:layout_constraintStart_toStartOf="@id/button"
    app:layout_constraintTop_toBottomOf="@id/button" />

</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/center_align_cl.xml)

The top `Button` centers itself within the `ConstraintLayout` and anchors itself to the top of the `ConstraintLayout`.

The bottom `Button` uses `app:layout_constraintStart_toStartOf` and `app:layout_constraintEnd_toEndOf` to align its edges with those of the top `Button`. However, its width is set to `wrap_content`, so the *size* of the `Button` will not change. Instead, those rules are treated as meaning that the bottom `Button` should be centered on the top `Button`.

In essence, both `Button` widgets are doing the same thing: using `app:layout_constraintStart_toStartOf` and `app:layout_constraintEnd_toEndOf` to center themselves. The difference is that the top `Button` is centering itself inside the `ConstraintLayout`, while the bottom `Button` is centering itself on the top `Button`.

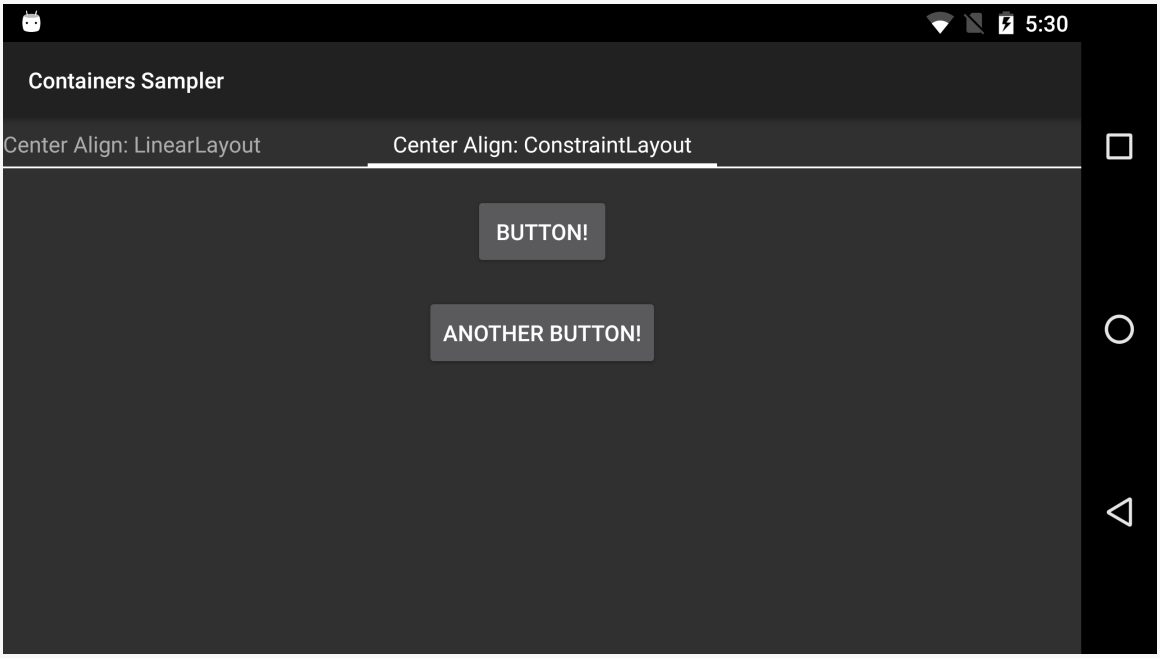


Figure 454: Center Alignment, Using ConstraintLayout

Assuming that we are putting the `ConstraintLayout` to other good uses, managing the position and size of other widgets, we are avoiding having a dedicated container for these two widgets. This has the potential to be more efficient and more flexible.

Keeping Things Proportional

Occasionally, we have a widget that we want to have a particular aspect ratio. That is relatively unusual. Perhaps the most common case is an `ImageView`, where you know in advance what the aspect ratio of the image is and you want to have the `ImageView` be sized to match.

`ConstraintLayout` offers an `app:layout_constraintDimensionRatio` that can help here.

To use the dimension ratio constraint, you need to pick one axis to have a known size (e.g., `wrap_content`). The other axis needs a size of `0dp`. Then, `app:layout_constraintDimensionRatio` will scale the widget along the `0dp` axis to match the specified aspect ratio:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
```

```
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ImageView
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:src="@drawable/freedom_tower"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintDimensionRatio="4:3"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

(from Containers/Sampler/app/src/main/res/layout/aspect_cl.xml)

Alternatively, you could have *both* axes have a size of `0dp`, and use an `H,` or `W,` prefix on the ratio to indicate which axis is the one that should be constrained (e.g., `W,4:3`).

Constraining the ConstraintLayout Size

Sometimes, your `ConstraintLayout` dimensions (`android:layout_width` and/or `android:layout_height`) are set to specific dimensions, ideally via a dimension resource. Sometimes, they are set to `match_parent`. And, sometimes, they are set to `wrap_content`. Those all work as expected.

However, sometimes, you would like `wrap_content` behavior, but only to certain extents. You want the `ConstraintLayout` to size itself based on its contents, but:

- You want to cap its size, perhaps to help prevent adjacent content from being clipped by the screen edge or being squished into too small of a space, or
- You want to prevent it from being *too* small, if the contents are small, because other aspects of the `ConstraintLayout` (e.g., background) are important and might not be visible at small size

To that end, `ConstraintLayout` offers the following attributes:

If This Is wrap_content...	...Then You Can Use
android:layout_width	android:minWidth and android:maxWidth
android:layout_height	android:minHeight and android:maxHeight

Each takes a dimension and works pretty much as you might expect from the attribute names.

Chains, Without the Whips

Chains represent a linked series of widgets in a row or column inside of a `ConstraintLayout`. You can then set up some rules for how the entire chain should be rendered, such as how to handle whitespace between those widgets.

How Do We Set up a Chain?

A chain is an implicit structure. There is no `app:chainId` or similar mechanism to identify chains.

Rather, a chain is defined any time there are 2 widgets with mutual constraints, such as the left side of one widget being tied to the right side of the other widget, and vice-versa.

A chain itself can be as many widgets as needed, so long as pair-wise they have the mutual constraints. So, for example, here is a chain containing three `Button` widgets:

```
<Button
  android:id="@+id/button1a"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="1"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintHorizontal_chainStyle="spread"
  app:layout_constraintLeft_toLeftOf="parent"
  app:layout_constraintRight_toLeftOf="@+id/button2a"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintVertical_bias="0.1" />

<Button
  android:id="@+id/button2a"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="2"
```

```
    app:layout_constraintBaseline_toBaselineOf="@+id/button1a"
    app:layout_constraintLeft_toRightOf="@+id/button1a"
    app:layout_constraintRight_toLeftOf="@+id/button3a" />

  <Button
    android:id="@+id/button3a"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="3"
    app:layout_constraintBaseline_toBaselineOf="@+id/button2a"
    app:layout_constraintLeft_toRightOf="@+id/button2a"
    app:layout_constraintRight_toRightOf="parent" />
```

(from Containers/Sampler/app/src/main/res/layout/chains_cl.xml)

`button1a` and `button2a` have their horizontal constraints connected to each other, as do `button2a` and `button3a`. Hence, the three widgets represent a single chain.

Styles of Chains

Configuring a chain comes in part by adding a "chain style" attribute to the head of the chain. The head is defined as the left-most widget in a horizontal chain (e.g., `button1a`) or the top-most widget in a vertical chain.

Spread

If you do not otherwise specify on a chain, the chain style is "spread". By default, this means that whitespace should be allocated between the widgets in the chain and outside the ends of the chain. And, by default, that allocation is even, so the space between each widget in the chain is the same, and the same amount of space appears outside the chain on either side.

That's what we get from the chain shown above:

Figure 455: Spread Chain

The chain head (`button1a`) has `app:layout_constraintHorizontal_chainStyle="spread"` to positively declare that

this chain has a spread style, though this is unnecessary, as this is the default chain behavior.

Spread-Inside

A `spread_inside` chain style is similar to `spread`, except that there is no whitespace added to either end of the chain. All whitespace goes between the pairs of widgets.

Figure 456: Spread-Inside Chain

Packed

Conversely, a `packed` chain style puts all the whitespace on either end of the chain, having the widgets directly adjacent to each other:

Figure 457: Packed Chain

Chains and Biases

Spread and packed chains — ones where there is whitespace outside the chain – respond to biases, where a bias slides the entire chain to a given side by allocating the whitespace non-equally.

For example, here is a biased packed chain:

```
<Button
  android:id="@+id/button1e"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="1"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintHorizontal_bias="0.8"
  app:layout_constraintHorizontal_chainStyle="packed"
  app:layout_constraintLeft_toLeftOf="parent"
  app:layout_constraintRight_toLeftOf="@+id/button2e"
  app:layout_constraintStart_toStartOf="parent"
```

```
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.9" />

  <Button
    android:id="@+id/button2e"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="2"
    app:layout_constraintBaseline_toBaselineOf="@+id/button1e"
    app:layout_constraintLeft_toRightOf="@+id/button1e"
    app:layout_constraintRight_toLeftOf="@+id/button3e" />

  <Button
    android:id="@+id/button3e"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="3"
    app:layout_constraintBaseline_toBaselineOf="@+id/button2e"
    app:layout_constraintLeft_toRightOf="@+id/button2e"
    app:layout_constraintRight_toRightOf="parent" />
```

(from Containers/Sampler/app/src/main/res/layout/chains_cl.xml)

Here, the chain head has `app:layout_constraintHorizontal_bias="0.8"`, sliding the entire packed chain to the right:

Figure 458: Packed Chain with Horizontal Bias

Chains and Weights

Spread and spread-inside chains — ones where there is whitespace inside the chain — respond to weights, reminiscent of the weights used with `LinearLayout`.

Simply marking the widget as having a width (or height, for vertical chains) of `0dp` causes that widget to absorb the whitespace:

```
  <Button
    android:id="@+id/button1c"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="1"
```

```
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintHorizontal_chainStyle="spread_inside"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toLeftOf="@+id/button2c"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.5" />

  <Button
    android:id="@+id/button2c"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="2"
    app:layout_constraintBaseline_toBaselineOf="@+id/button1c"
    app:layout_constraintLeft_toRightOf="@+id/button1c"
    app:layout_constraintRight_toLeftOf="@+id/button3c" />

  <Button
    android:id="@+id/button3c"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="3"
    app:layout_constraintBaseline_toBaselineOf="@+id/button2c"
    app:layout_constraintLeft_toRightOf="@+id/button2c"
    app:layout_constraintRight_toRightOf="parent" />
```

(from Containers/Sampler/app/src/main/res/layout/chains_cl.xml)

Figure 459: Spread-Inside Chain, With Middle Button Filling Space

If two or more widgets use `0dp` this way, they will divide the space equally, by default. Using `app:layout_constraintHorizontal_weight` or `app:layout_constraintVertical_weight` on the affected widgets can be used to divide the space on a weighted basis (e.g., giving one widget twice the space of another).

What If We're GONE?

For most container classes, marking a widget's visibility as `GONE` totally removes it from everything regarding the measurement and layout of the container. It is as if

the widget never existed, though you are in position to "add it back in" simply by making it `VISIBLE`.

`ConstraintLayout` handles this slightly differently. Widgets that are `GONE` are treated as having no size (width and height of 0) and having no margins. However, `ConstraintLayout` will still honor constraints tied to those `GONE` widgets, taking into account their zero-size and zero-margin status. This allows your constraints to still work, rather than breaking entirely.

The zero size is non-negotiable. However, it is possible that you may still want margins to be taken into account. To that end, there are attributes like `app:layout_goneMarginStart` that you can use. As the attribute name suggests, these provide margins to be used when the widget itself is marked as `GONE`. You could set these to the same value as you used for the corresponding normal margin, perhaps using the data binding framework to have one attribute value reference the other attribute value and reduce duplication. You could even set the `GONE` margins to be something entirely different, if that made sense in your scenario (e.g., half of the normal margin).

GridLayout

In 2011, Google added `GridLayout` to our roster of available container classes (a.k.a., layout managers). `GridLayout` is an attempt to make setting up complex Android layouts a bit easier, particularly with an eye towards working well with IDE graphical layout editors. In this chapter, we will examine why `GridLayout` was added and how we can use it in our projects.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Issues with the Classic Containers

Most layouts are implemented using a combination of `LinearLayout`, `RelativeLayout`, and `TableLayout`. Almost everything you would want to be able to create can be accomplished using one, or sometimes more than one, of those containers.

However, there are issues with the classic containers. The two most prominent might be the over-reliance upon nested containers and issues with drag-and-drop GUI building capability.

Nested Containers

`LinearLayout` and `TableLayout` suffer from a tendency to put too many containers inside of other containers. For example, implementing some sort of 2x2 grid would involve:

- A vertical `LinearLayout` holding onto a pair of horizontal `LinearLayouts`, or
- A `TableLayout` holding onto a pair of `TableRows`

On the surface, this does not seem that bad. And, in many cases, it is not that bad.

However, views and containers are relatively heavyweight items. They consume a fair bit of heap space, and when it comes time to lay them out on the screen, they consume a fair bit of processing power. In particular, the fact that a container can hold onto *any* type of widget or container means that it is difficult to optimize common scenarios (e.g., a 2x2 grid) for faster processing. Instead, a container treats its children more or less as "black boxes", requiring lots of method invocations up and down the call stack to calculate sizes and complete the layout process.

Moreover, the call stack itself can be an issue. The stack size of the main application thread has historically been rather small (8KB was the last reported value). If you have a complex UI, with more than ~15 nested containers, you are likely to run into a `StackOverflowError`. Android itself will contribute some of these containers, exacerbating this problem.

`RelativeLayout`, by comparison, can implement some UI patterns without any nested containers, simply by positioning widgets relative to the container's bounds and relative to each other.

Drag-and-Drop

Where `RelativeLayout` falls down is with the drag-and-drop capability of the graphical layout editor in IDEs like Android Studio.

When you release the mouse button when dropping a widget into the preview area, the tools need to determine what that really means in terms of layout rules.

`LinearLayout` works fairly well: it will either insert your widget in between two other widgets or add it to the end of the row or column you dropped into. `TableLayout` behaves similarly.

`RelativeLayout`, though, has a more difficult time guessing what particular combination of rules you really mean by this particular drop target. Are you trying to attach the widget to another widget? If so, which one? Are you trying to attach the widget to the bounds of the `RelativeLayout`? While sometimes it will guess properly, sometimes it will not, with potentially confusing results. It is reasonably

likely that you will need to tweak the layout rules manually, either via the Attributes pane or via the raw XML.

The New Contender: GridLayout

`GridLayout` tries to cull the best of the capabilities of the classic containers and drop as many of their limitations as possible.

`GridLayout` works a bit like `TableLayout`, insofar as it sets things up in a grid, with rows and columns, where the row and column sizes are computed based upon what is placed into those rows and columns. However, unlike `TableLayout`, which relies upon a separate `TableRow` container to manage the rows, `GridLayout` takes the `RelativeLayout` approach of putting rules on the individual widgets (or containers) in the grid, where those rules steer the layout processing. For example, with `GridLayout`, widgets can declare specifically which row and column they should slot into.

`GridLayout` also goes a bit beyond what `TableLayout` offers in terms of capabilities. Notably, it supports row spans as well as column spans, whereas `TableRow` only supports a column span. This gives you greater flexibility when designing your layout to fit the grid-style positioning rules. You can also:

- Explicitly state how many columns there are, rather than having that value be inferred by row contents
- Allow Android to determine where to place a widget without specifying any row or column, with it finding the next available set of grid cells capable of holding the widget, based upon its requested row span and column span values
- Have control over orientation: whereas `TableLayout` always was a column of rows, you could have a `GridLayout` be a row of columns, if that makes implementing the design easier
- And so on

GridLayout and the Android Support Package

`GridLayout` was natively added to the Android SDK in API Level 14 (Android 4.0). Fortunately, the Android Support package has a backport of `GridLayout`. However, the backport is not in one of the JAR files, such as `support-v4`, as `GridLayout` requires some resources. Hence, it is in an Android library project that you must add to your project, known as `gridlayout-v7`.

Android Studio users can simply add a `implementation 'com.android.support:gridlayout-v7:...'` statement to their top-level `dependencies` closure, for some version identified by `...`. So long as those users have the Android Support Repository set up in the SDK Manager, Gradle will be able to find and incorporate the artifact.

When using the backported `GridLayout`, you will need to declare another XML namespace in your layout XML resources. That namespace will be `http://schemas.android.com/apk/res-auto`. If you use an IDE to add the `GridLayout` to the layout resource, it will automatically add this namespace, under the prefix of `app`, such as:

```
<android.support.v7.widget.GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">
</android.support.v7.widget.GridLayout>
```

That namespace is required for `GridLayout`-specific attributes. For example, we can have a `columnCount` attribute, indicating how many columns the `GridLayout` should contain. For the native API Level 14 `GridLayout`, that attribute would be `android:columnCount`. For the backport, it will be `app:columnCount`, assuming that you gave the namespace the prefix of `app`.

When citing `GridLayout`-specific attributes, the rest of this chapter will use the `app` prefix, to clarify which attributes need that prefix for the backport. If you are using the native API Level 14 implementation of `GridLayout`, and you are manually working with the XML, just remember to use `android` as a prefix instead of `app`.

The sample app shows *both* the native and the backport implementations of `GridLayout`: on API Level 14+ devices/emulators it will use native implementations from `res/layout-v14/`, and it will use the backport on older environments.

Our Test App

To look at a series of `GridLayout`-based layouts, let's turn our attention to the `GridLayout/Sampler` sample project. This has the same `ViewPager` and `PagerTabStrip` as did the second sample app from the chapter on `ViewPager`. However, rather than use a list of 10 `EditText` widgets managed by fragments, in this

case, our fragments will manage layouts containing `GridLayout`. Each page of our pager will contain a `TrivialFragment`, whose contents are based on a `Sample` class that is a simple pair of a layout resource ID and a string resource ID for the fragment's title:

```
package com.commonsware.android.gridlayout;

class Sample {
  int layoutId;
  int titleId;

  Sample(int layoutId, int titleId) {
    this.layoutId=layoutId;
    this.titleId=titleId;
  }
}
```

(from GridLayout/Sampler/app/src/main/java/com/commonsware/android/gridlayout/Sample.java)

Our revised `SampleAdapter` maintains a static `ArrayList` of these `Sample` objects, one per layout we wish to examine, and uses those values to populate our `ViewPager` title:

```
package com.commonsware.android.gridlayout;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;
import java.util.ArrayList;

public class SampleAdapter extends FragmentPagerAdapter {
  static ArrayList<Sample> SAMPLES=new ArrayList<Sample>();
  private Context ctxt=null;

  static {
    SAMPLES.add(new Sample(R.layout.row, R.string.row));
    SAMPLES.add(new Sample(R.layout.column, R.string.column));
    SAMPLES.add(new Sample(R.layout.table, R.string.table));
    SAMPLES.add(new Sample(R.layout.table_flex, R.string.flexible_table));
    SAMPLES.add(new Sample(R.layout.implicit, R.string.implicit));
    SAMPLES.add(new Sample(R.layout.spans, R.string.spans));
  }

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
```

```
  }

  @Override
  public int getCount() {
    return(SAMPLES.size());
  }

  @Override
  public Fragment getItem(int position) {
    return(TrivialFragment.newInstance(getSample(position).layoutId));
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(getSample(position).titleId));
  }

  private Sample getSample(int position) {
    return(SAMPLES.get(position));
  }
}
```

(from GridLayout/Sampler/app/src/main/java/com/commonsware/android/gridlayout/SampleAdapter.java)

`TrivialFragment` just inflates our desired layout, having received the layout resource ID as a parameter to its factory method:

```
package com.commonsware.android.gridlayout;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class TrivialFragment extends Fragment {
  private static final String KEY_LAYOUT_ID="layoutId";

  static TrivialFragment newInstance(int layoutId) {
    TrivialFragment frag=new TrivialFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_LAYOUT_ID, layoutId);
    frag.setArguments(args);

    return(frag);
  }
```

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    return(inflater.inflate(getArguments().getInt(KEY_LAYOUT_ID, -1),
                            container, false));
  }
}
```

(from GridLayout/Sampler/app/src/main/java/com/commonsware/android/gridlayout/TrivialFragment.java)

Note that if you load this project from the GitHub repository, you will need to update it for your copy of the `GridLayout` library project.

Replacing the Classics

Let's first examine the behavior of `GridLayout` by seeing how it can replace some of the classic layouts we would get from `LinearLayout` and `TableLayout`. Each of the following sub-sections will examine one `GridLayout`-based layout XML resource, how it can be constructed, and what the result looks like when viewed in the sample project.

Horizontal LinearLayout

The classic way to create a row of widgets is to use a horizontal `LinearLayout`. The `LinearLayout` will put each of its children, one after the next, within the row.

The `GridLayout` equivalent is to specify one that has an `app:columnCount` equal to the number of widgets in the row. Then, each widget will have `app:layout_column` set to its specific column index (starting at `0`) and `app:layout_row` set to `0`, as seen in `res/layout/row.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">

  <Button
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/button"/>

  <Button
    app:layout_column="1"
    app:layout_row="0"
    android:text="@string/button"/>
```

```
</android.support.v7.widget.GridLayout>
```

(from GridLayout/Sampler/app/src/main/res/layout/row.xml)

Unlike `LinearLayout`, though, we do not specify sizes of the children, in terms of `android:layout_width` and `android:layout_height`. `GridLayout` works a bit like `TableLayout` in this regard, supplying default values for these attributes. In the case of `GridLayout`, the defaults are `wrap_content`, and this cannot be overridden (akin to the behavior of immediate children of a `TableRow`). Instead, you will control size via row and column spans, as will be illustrated later in this chapter.

Given the above layout, we get:

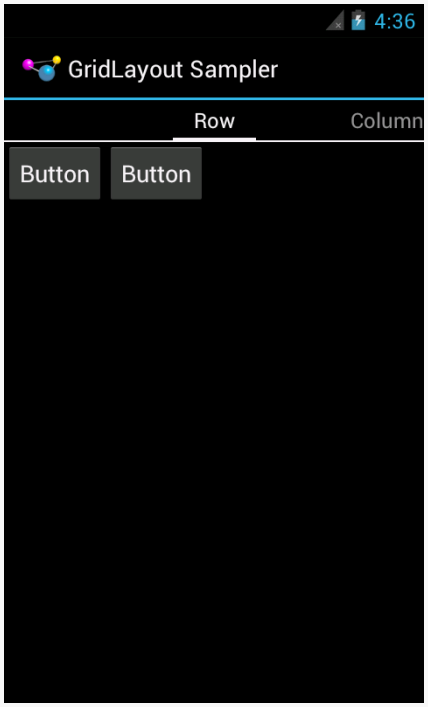


Figure 460: Row Using GridLayout, on a 4.0.3 Emulator

Vertical LinearLayout

Similarly, the conventional way you would specify a column is to use a vertical `LinearLayout`, which would position its children one after the next. The `GridLayout` equivalent would be to have `app:columnCount` set to `1`, and to place the widgets in each required row via `app:layout_row` attributes, as seen in `res/layout/column.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="1">

  <Button
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/button"/>

  <Button
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/button"/>

</android.support.v7.widget.GridLayout>
```

(from GridLayout/Sampler/app/src/main/res/layout/column.xml)

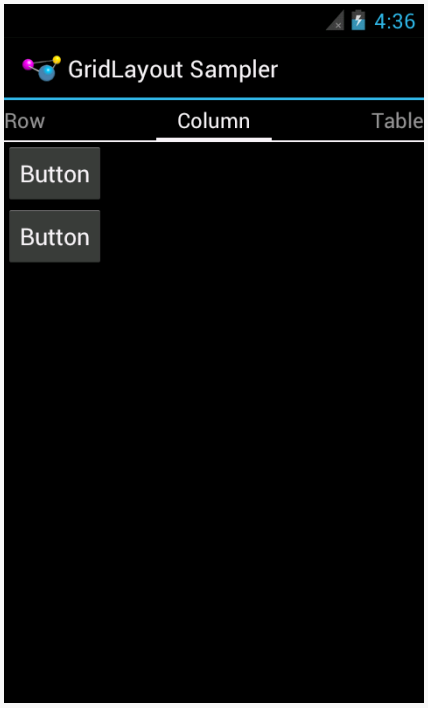


Figure 461: Column Using GridLayout, on a 4.0.3 Emulator

All that being said, it is still probably better to use `LinearLayout` in these cases, rather than mess with `GridLayout`.

TableLayout

The big key to a `TableLayout` is column width, where columns expand to fill their contents, assuming there is sufficient room in the table. `GridLayout` also expands its columns to address the sizes of its contents.

For example, here is a simple 2x2 table, with `TextView` widgets in the left column and `EditText` widgets in the right column, as seen in `res/layout/table.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">

  <TextView
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="0"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="1"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

(from GridLayout/Sampler/app/src/main/res/layout/table.xml)

However, our `EditText` widgets are small, because nothing is causing them to fill the available space. To do that, we can use `android:layout_gravity`, to ask the `GridLayout` to let the widgets fill the available horizontal space, as seen in `res/layout/table_flex.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
```

```
  app:columnCount="2">

  <TextView
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="0"
    app:layout_gravity="fill_horizontal"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="1"
    app:layout_gravity="fill_horizontal"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

(from GridLayout/Sampler/app/src/main/res/layout/table_flex.xml)

This allows the `EditText` widgets to fill the width of the column:

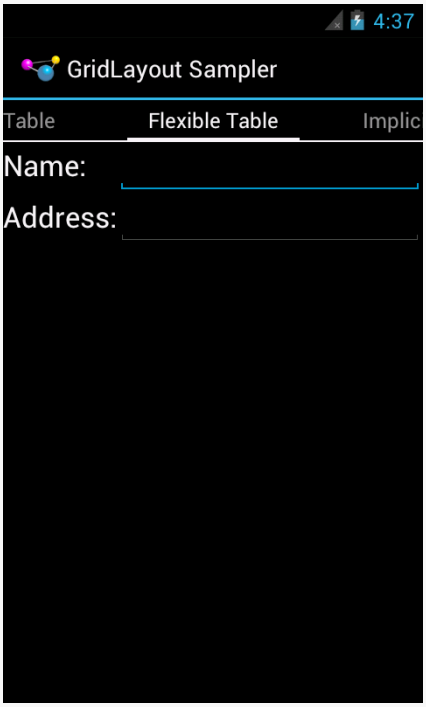


Figure 462: Table Using GridLayout, on a 4.0.3 Emulator

That holds true regardless of how wide that column is:

Figure 463: Table Using GridLayout, in Landscape, on a 4.0.3 Emulator

Implicit Rows and Columns

While all the previous samples showed the row and column of each widget being defined explicitly via `app:layout_row` and `app:layout_column` attributes, that is not your only option.

If you have `app:columnCount` on the `GridLayout` element itself, you can allow `GridLayout` to assign rows and columns. In this respect, `GridLayout` behaves a bit like a "flow layout": it assigns widgets to cells in the first row, starting from the first column and working its way across, wrapping to the next row when it runs out of room. This makes for a more terse layout file, at the cost of perhaps introducing a bit of confusion when you add or remove a widget and everything after it in the layout file shifts location.

For example, `res/layout/implicit.xml` is the same as `res/layout/table_flex.xml`, except that it skips the `app:layout_row` and `app:layout_column` attributes, allowing `GridLayout` to assign the positions:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2"
  app:orientation="horizontal">

  <TextView
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_gravity="fill_horizontal"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_gravity="fill_horizontal"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

(from GridLayout/Sampler/app/src/main/res/layout/implicit.xml)

Visually, this sample is identical to the last one:

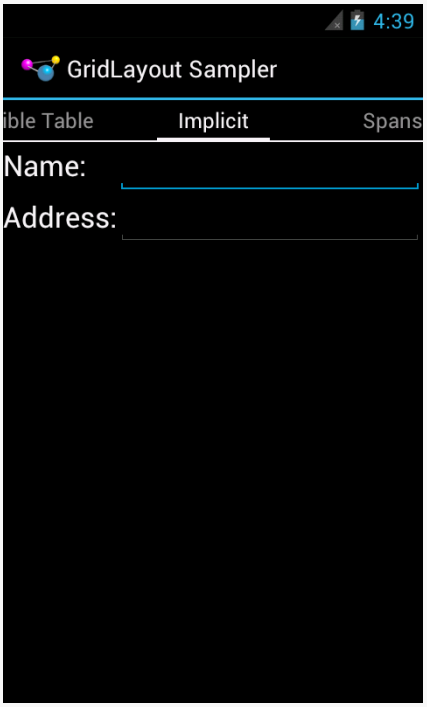


Figure 464: Table Using GridLayout and Implicit Positions, on a 4.0.3 Emulator

The "across columns, then down rows" model holds for `GridLayout` in the default orientation: horizontal. You can add an `app:orientation` attribute to the `GridLayout`, setting it to `vertical`. Then, based on an `app:rowCount` value, `GridLayout` will automatically assign positions, working down the first column, then across to the next column when it runs out of rows.

Row and Column Spans

Like `TableLayout`, `GridLayout` supports the notion of column spans. You can use `app:layout_columnSpan` to indicate how many columns a particular widget should span in the resulting grid.

However, `GridLayout` also supports row spans, in the form of `app:layout_rowSpan` attributes. A widget can span rows, columns, or both, as needed.

If you are using implicit positions, per the previous section, `GridLayout` will seek the next available space that has sufficient rows and columns for a widget's set of spans.

For example, the following diagram depicts five buttons placed in a `GridLayout` with various spans, and an attempt to add a sixth button that should span two columns:

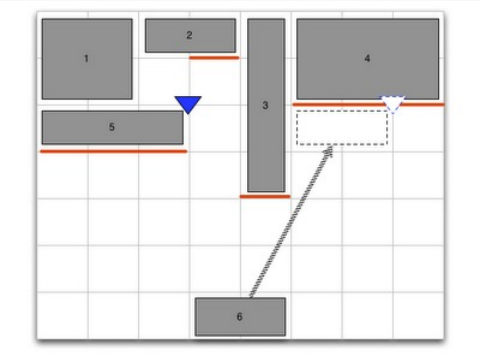


Figure 465: Span Sample (image courtesy of Android Open Source Project)

Assuming the first five buttons were added in sequence and with implicit positioning, `GridLayout` ordinarily would drop the sixth button into the fourth column of the third row. However, there is only a one-column-wide space available there, given that the third button intrudes into the third row. Hence, `GridLayout` will skip over the smaller space and put the sixth button into the sixth column in the third row.

A `GridLayout`-based layout that implements the above diagram can be found in `res/layout/spans.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="9"
  app:orientation="horizontal"
  app:rowCount="5">

  <Button
    app:layout_gravity="fill"
    app:layout_columnSpan="2"
    app:layout_rowSpan="2"
    android:text="@string/string_1"/>

  <Button
    app:layout_gravity="fill_horizontal"
    app:layout_columnSpan="2"
    android:text="@string/string_2"/>

  <Button
    app:layout_gravity="fill_vertical"
    app:layout_rowSpan="4"
    android:text="@string/string_3"/>
```

```
<Button
  app:layout_gravity="fill"
  app:layout_columnSpan="3"
  app:layout_rowSpan="2"
  android:text="@string/string_4"/>

<Button
  app:layout_gravity="fill_horizontal"
  app:layout_columnSpan="3"
  android:text="@string/string_5"/>

<Button
  app:layout_gravity="fill_horizontal"
  app:layout_columnSpan="2"
  android:text="@string/string_6"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="0"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="1"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="2"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="3"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="4"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="5"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="6"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="7"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_height="36dp"
  app:layout_column="8"
```

```
    app:layout_row="0" />

  <android.support.v7.widget.Space
    android:layout_height="36dp"
    app:layout_column="8"
    app:layout_row="1" />

  <android.support.v7.widget.Space
    android:layout_height="36dp"
    app:layout_column="8"
    app:layout_row="2" />

  <android.support.v7.widget.Space
    android:layout_height="36dp"
    app:layout_column="8"
    app:layout_row="3" />

  <android.support.v7.widget.Space
    android:layout_height="36dp"
    app:layout_column="8"
    app:layout_row="4" />

</android.support.v7.widget.GridLayout>
```

(from GridLayout/Sampler/app/src/main/res/layout/spans.xml)

This layout shows one of the limitations of `GridLayout`: its columns and rows will have a size of `0` by default. Hence, to ensure that each row and column has a minimum size, this layout uses `Space` elements (in an eighth column and fifth row) to establish those minimums. This makes the layout file fairly verbose, but it gives the desired results:

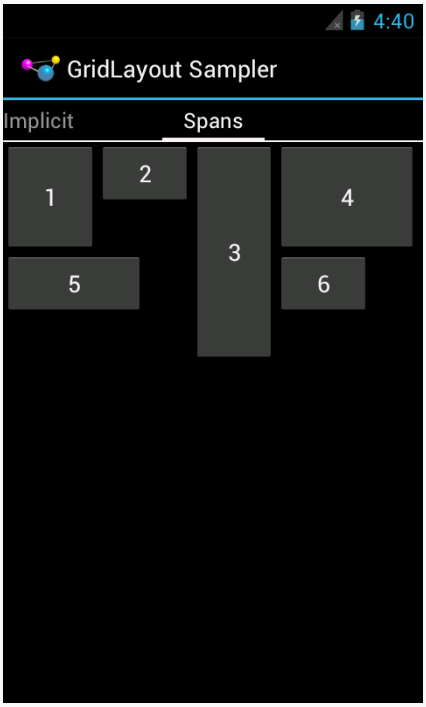


Figure 466: GridLayout Spans, on a 4.0.3 Emulator

However, the fixed-sized `Space` elements break the fluidity of the layout:

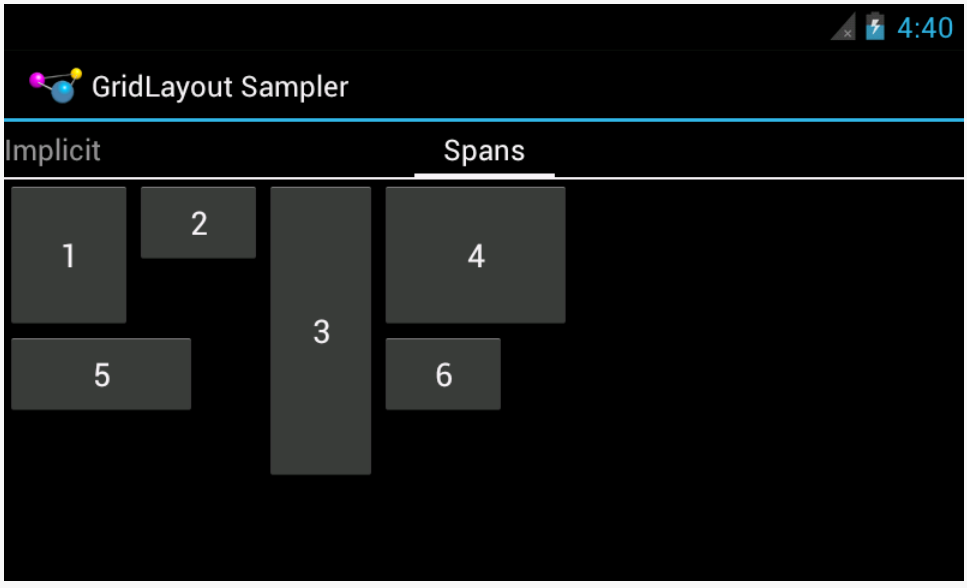


Figure 467: GridLayout Spans, in Landscape, on a 4.0.3 Emulator

Perhaps someday someone will create a `PercentSpace` widget, occupying a percentage of the parent's size, that could be used instead.

The author would like to give thanks to those on Stack Overflow who assisted in getting the span layout to work.

Dialogs and DialogFragments

Generally speaking, modal dialogs are considered to offer poor UX, particularly on mobile devices. You want to give the user more choices, not fewer, and so locking them into "deal with this dialog right now, or else" is not especially friendly. That being said, from time to time, there will be cases where that sort of modal interface is necessary, and to help with that, Android does have a dialog framework that you can use.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

DatePickerDialog and TimePickerDialog

Android has a pair of built-in dialogs that handle the common operations of allowing the user to select a date (`DatePickerDialog`) or a time (`TimePickerDialog`). These are simply dialog wrappers around the `DatePicker` and `TimePicker` widgets, as are described in this book's Widget Catalog.

The `DatePickerDialog` allows you to set the starting date for the selection, in the form of a year, month, and day of month value. Note that the month runs from `0` for January through `11` for December. Most importantly, both let you provide a callback object (`OnDateChangedListener` or `OnDateSetListener`) where you are informed of a new date selected by the user. It is up to you to store that date someplace, particularly if you are using the dialog, since there is no other way for you to get at the chosen date later on.

Similarly, `TimePickerDialog` lets you:

- Set the initial time the user can adjust, in the form of an hour (`0` through `23`) and a minute (`0` through `59`)
- Indicate if the selection should be in 12-hour mode with an AM/PM toggle, or in 24-hour mode (what in the US is thought of as "military time" and what in much of the rest of the world is thought of as "the way times are supposed to be")
- Provide a callback object (`OnTimeChangedListener` or `OnTimeSetListener`) to be notified of when the user has chosen a new time, which is supplied to you in the form of an hour and minute

For example, from the Dialogs/Chrono sample project, here's a trivial layout containing a label and two buttons — the buttons will pop up the dialog flavors of the date and time pickers:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView android:id="@+id/dateAndTime"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <Button android:id="@+id/dateBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Set the Date"
    android:onClick="chooseDate"
    />
  <Button android:id="@+id/timeBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Set the Time"
    android:onClick="chooseTime"
    />
</LinearLayout>
```

(from Dialogs/Chrono/app/src/main/res/layout/main.xml)

The more interesting stuff comes in the Java source:

```
package com.commonsware.android.chrono;

import android.app.Activity;
import android.app.DatePickerDialog;
import android.app.TimePickerDialog;
import android.os.Bundle;
import android.text.format.DateUtils;
import android.view.View;
import android.widget.DatePicker;
import android.widget.TextView;
import android.widget.TimePicker;
import java.util.Calendar;

public class ChronoDemo extends Activity {
  TextView dateAndTimeLabel;
  Calendar dateAndTime=Calendar.getInstance();

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    dateAndTimeLabel=(TextView)findViewById(R.id.dateAndTime);

    updateLabel();
  }

  public void chooseDate(View v) {
    new DatePickerDialog(this, d,
                         dateAndTime.get(Calendar.YEAR),
                         dateAndTime.get(Calendar.MONTH),
                         dateAndTime.get(Calendar.DAY_OF_MONTH))
      .show();
  }

  public void chooseTime(View v) {
    new TimePickerDialog(this, t,
                         dateAndTime.get(Calendar.HOUR_OF_DAY),
                         dateAndTime.get(Calendar.MINUTE),
                         true)
      .show();
  }

  private void updateLabel() {
    dateAndTimeLabel
      .setText(DateUtils
               .formatDateTime(this,
                               dateAndTime.getTimeInMillis(),
                               DateUtils.FORMAT_SHOW_DATE|DateUtils.FORMAT_SHOW_TIME));
  }

  DatePickerDialog.OnDateSetListener d=new DatePickerDialog.OnDateSetListener() {
    public void onDateSet(DatePicker view, int year, int monthOfYear,
                          int dayOfMonth) {
      dateAndTime.set(Calendar.YEAR, year);
      dateAndTime.set(Calendar.MONTH, monthOfYear);
      dateAndTime.set(Calendar.DAY_OF_MONTH, dayOfMonth);
      updateLabel();
    }
  };
```

```
  TimePickerDialog.OnTimeSetListener t=new TimePickerDialog.OnTimeSetListener() {
    public void onTimeSet(TimePicker view, int hourOfDay,
                          int minute) {
      dateAndTime.set(Calendar.HOUR_OF_DAY, hourOfDay);
      dateAndTime.set(Calendar.MINUTE, minute);
      updateLabel();
    }
  };
}
```

(from Dialogs/Chrono/app/src/main/java/com/commonsware/android/chrono/ChronoDemo.java)

The "model" for this activity is just a `Calendar` instance, initially set to be the current date and time. In the `updateLabel()` method, we take the current `Calendar`, format it using `DateUtils` and `formatDateTime()`, and put it in the `TextView`. The nice thing about using Android's `DateUtils` class is that it will format dates and times using the user's choice of date formatting, determined through the Settings application.

Each button has a corresponding method that will get control when the user clicks it (`chooseDate()` and `chooseTime()`). When the button is clicked, either a `DatePickerDialog` or a `TimePickerDialog` is shown. In the case of the `DatePickerDialog`, we give it an `OnDateSetListener` callback that updates the `Calendar` with the new date (year, month, day of month). We also give the dialog the last-selected date, getting the values out of the `Calendar`. In the case of the `TimePickerDialog`, it gets an `OnTimeSetListener` callback to update the time portion of the `Calendar`, the last-selected time, and a true indicating we want 24-hour mode on the time selector

With all this wired together, the resulting activity looks like this:

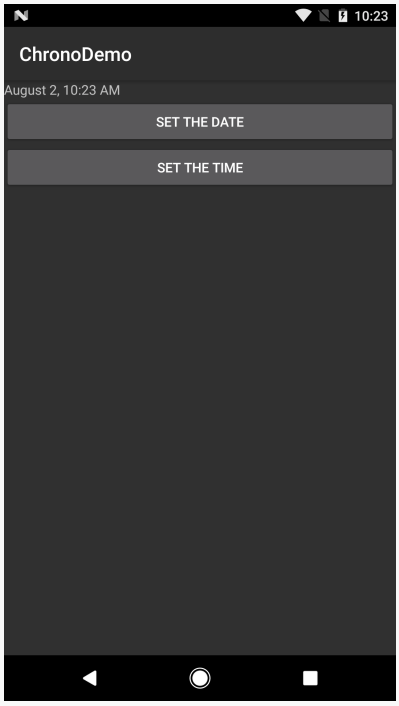


Figure 468: ChronoDemo, As Initially Launched, on Android 7.1

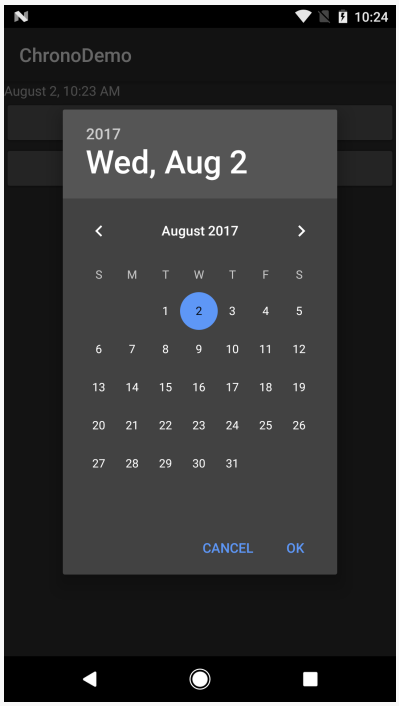


Figure 469: ChronoDemo, Showing DatePickerDialog

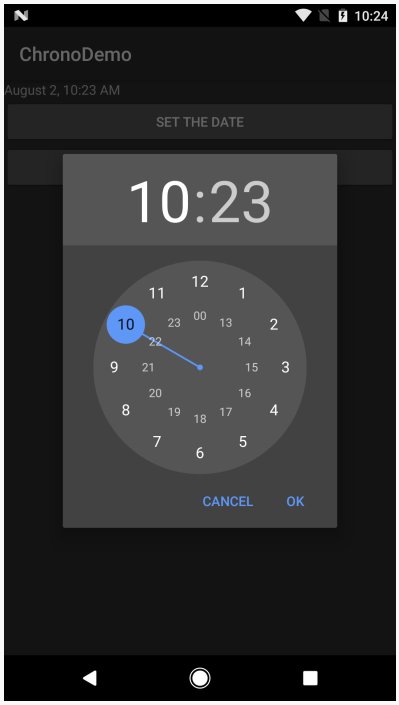


Figure 470: ChronoDemo, Showing TimePickerDialog

Changes and Bugs

Android 4.1 through 4.4 have some changes in behavior from what came before and what came after.

First, the "Cancel" button was removed, unless you specifically add a negative button listener to the underlying `DatePicker` or `TimePicker` widget:

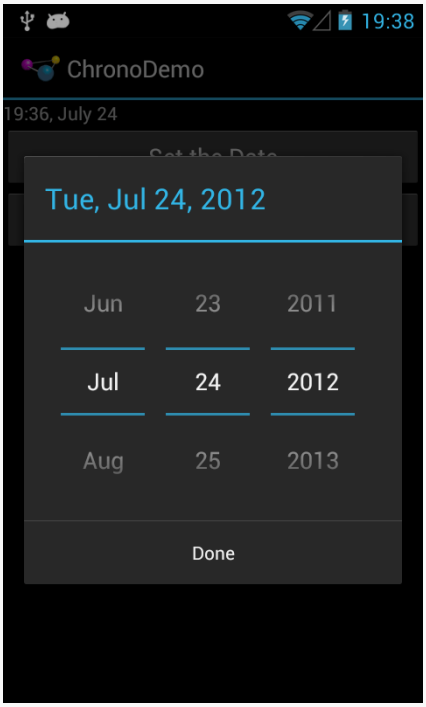


Figure 471: ChronoDemo, Showing DatePickerDialog, on Android 4.1

The user can press BACK to exit the dialog, so all functionality is still there, but you may need to craft your documentation to accommodate this difference. And, on Android 5.0+, the Cancel button returned.

Second, your `OnDateSetListener` or `OnTimeSetListener` will be called an extra time. If the user presses BACK to leave the dialog, your `onDateSet()` or `onTimeSet()` will be called. If the user clicks the positive button of the dialog, you are called *twice*. There is a workaround documented on Stack Overflow. This too was repaired in Android 5.0.

AlertDialog

For your own custom dialogs, you could extend the `Dialog` base class, as do `DatePickerDialog` and `TimePickerDialog`. More commonly, though, developers create custom dialogs via `AlertDialog`, in large part due to the existence of `AlertDialog.Builder`. This builder class allows you to construct a custom dialog using a single (albeit long) Java statement, rather than having to create your own custom subclass. `Builder` offers a series of methods to configure an `AlertDialog`, each method returning the `Builder` for easy chaining.

Commonly-used configuration methods on Builder include:

- `setMessage()` if you want the "body" of the dialog to be a simple textual message, from either a supplied String or a supplied string resource ID.
- `setTitle()` and `setIcon()`, to configure the text and/or icon to appear in the title bar of the dialog box.
- `setPositiveButton()`, `setNeutralButton()`, and `setNegativeButton()`, to indicate which button(s) should appear across the bottom of the dialog, where they should be positioned (left, center, or right, respectively), what their captions should be, and what logic should be invoked when the button is clicked (besides dismissing the dialog).

Calling `create()` on the `Builder` will give you the `AlertDialog`, built according to your specifications. You can use additional methods on `AlertDialog` itself to perhaps configure things beyond what `Builder` happens to support.

Note, though, that calling `create()` does not actually display the dialog. The modern way to display the dialog is to tie it to a `DialogFragment`, as will be discussed in the next section.

DialogFragments

One challenge with dialogs comes with configuration changes, notably screen rotations. If they pivot the device from portrait to landscape (or vice versa), presumably the dialog should remain on the screen after the change. However, since Android wants to destroy and recreate the activity, that would have dire impacts on your dialog.

Pre-fragments, Android had a "managed dialog" facility that would attempt to help with this. However, with the introduction of fragments came the `DialogFragment`, which handles the configuration change process.

You have two ways of supplying the dialog to the `DialogFragment`:

1. You can override `onCreateDialog()` and return a `Dialog`, such as `AlertDialog` created via an `AlertDialog.Builder`
2. You can override `onCreateView()`, as you would with an ordinary fragment, and the `View` that you return will be placed inside of a dialog

The Dialogs/DialogFragment sample project demonstrates the use of a `DialogFragment` in conjunction with an `AlertDialog` in this fashion.

Here is our `DialogFragment`, named `SampleDialogFragment`:

```
package com.commonsware.android.dlgfrag;

import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.content.DialogInterface;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class SampleDialogFragment extends DialogFragment implements
    DialogInterface.OnClickListener {
  private View form=null;

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    form=
        getActivity().getLayoutInflater()
                     .inflate(R.layout.dialog, null);

    AlertDialog.Builder builder=new AlertDialog.Builder(getActivity());

    return(builder.setTitle(R.string.dlg_title).setView(form)
                  .setPositiveButton(android.R.string.ok, this)
                  .setNegativeButton(android.R.string.cancel, null).create());
  }

  @Override
  public void onClick(DialogInterface dialog, int which) {
    String template=getActivity().getString(R.string.toast);
    EditText name=(EditText)form.findViewById(R.id.title);
    EditText value=(EditText)form.findViewById(R.id.value);
    String msg=
        String.format(template, name.getText().toString(),
                      value.getText().toString());

    Toast.makeText(getActivity(), msg, Toast.LENGTH_LONG).show();
  }

  @Override
```

```
  public void onDismiss(DialogInterface unused) {
    super.onDismiss(unused);

    Log.d(getClass().getSimpleName(), "Goodbye!");
  }

  @Override
  public void onCancel(DialogInterface unused) {
    super.onCancel(unused);

    Toast.makeText(getActivity(), R.string.back, Toast.LENGTH_LONG).show();
  }
}
```

(from Dialogs/DialogFragment/app/src/main/java/com/commonsware/android/dlgfrag/SampleDialogFragment.java)

In `onCreateDialog()`, we inflate a custom layout (`R.layout.dialog`) that consists of some `TextView` labels and `EditText` fields:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_name"/>

    <EditText
      android:id="@+id/title"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text"/>
  </LinearLayout>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:orientation="horizontal">
```

```
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/value"/>

    <EditText
      android:id="@+id/value"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="number"/>
  </LinearLayout>

</LinearLayout>
```

(from Dialogs/DialogFragment/app/src/main/res/layout/dialog.xml)

We then create an instance of `AlertDialog.Builder`, then start configuring the dialog by calling a series of methods on the `Builder`:

- `setTitle()` to supply the text to appear in the title bar of the dialog
- `setView()` to define the contents of the dialog, in the form of our inflated `View`
- `setPositiveButton()` to define the caption of one button (set here to the Android-supplied "OK" string resource) and to arrange to get control when that button is clicked (via `this` as the second parameter and our activity implementing `DialogInterface.OnClickListener`)
- `setNegativeButton()` to define the caption of the other button (set here to the Android-supplied "Cancel" resource)

We do not supply a listener to `setNegativeButton()`, because we do not need one in this case. Whenever the user clicks on *any* of the buttons, the dialog will be dismissed automatically. Hence, you only need a listener if you intend to do something special beyond dismissing the dialog when a button is clicked.

At that point, we call `create()` to construct the actual `AlertDialog` instance and hand that back to Android.

If the user taps our positive button, we are called with `onClick()` and can collect information from our form and do something with it, in this case displaying a `Toast`.

We also override:

- `onCancel()`, which is called if the user presses the BACK button to exit the dialog
- `onDismiss()`, which is called whenever the dialog goes away for any reason (BACK or a button click)

Our activity (`MainActivity`), has a big button tied to a `showMe()` method, which calls `show()` on a newly-created instance of our `SampleDialogFragment`:

```
public void showMe(View v) {
  new SampleDialogFragment().show(getFragmentManager(), "sample");
}
```

(from Dialogs/DialogFragment/app/src/main/java/com/commonsware/android/dlgfrag/MainActivity.java)

The second parameter to `show()` is a tag that can be used to retrieve this fragment again later from the `FragmentManager` via `findFragmentByTag()`.

When you click the big button in the activity, our dialog is displayed:

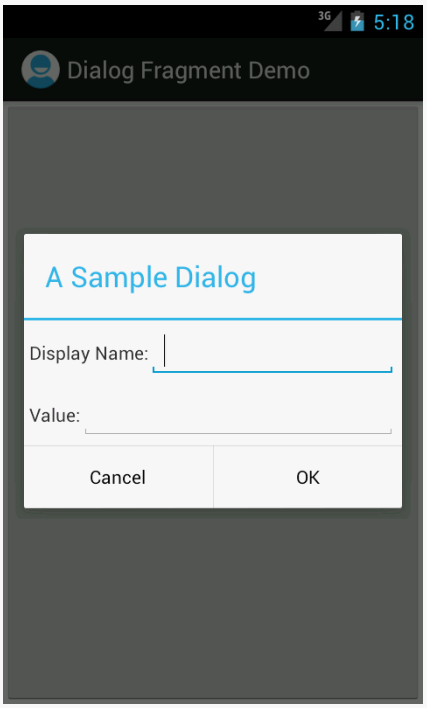


Figure 472: SampleDialogFragment, As Initially Launched, on Android 4.0.3

Android will handle the configuration change, and so long as our dialog uses typical widgets like `EditText`, the standard configuration change logic will carry our data forward from the old activity's dialog to the new activity's dialog.

DialogFragment: The Other Flavor

If you do not override `onCreateDialog()`, Android will assume that you want the `View` returned by `onCreateView()` to be poured into an ordinary `Dialog`, which `DialogFragment` will create for you automatically.

One advantage of this approach is that you can selectively show the fragment as a dialog *or* show it as a regular fragment as part of your main UI.

To show the fragment as a dialog, use the same `show()` technique as was outlined in the previous section. To display the fragment as part of the main UI, use a `FragmentTransaction` to `add()` it, the way you would for any other dynamic fragment.

This is one alternative to the normal fragment approach of having dedicated activities for each fragment on smaller screen sizes.

We will also see this approach used when we try to apply fragments to display content on a secondary screen using Android 4.2's `Presentation` class, covered elsewhere in this book.

Dialogs: Modal, Not Blocking

Dialogs in Android are modal in terms of UI. The user cannot proceed in your activity until they complete or dismiss the dialog.

Dialogs in Android are not blocking in terms of the programming model. When you call `show()` to display a dialog — either directly or by means of adding a `DialogFragment` to the screen — this is not a blocking call. The dialog will be displayed sometime after the call to `show()`, asynchronously. You use callbacks, such as the button event listeners, to find out about events going on with respect to the dialog that you care about.

This runs counter to a couple of GUI toolkits, where displaying the dialog blocks the thread that does the displaying. In those toolkits, the call to `show()` would not return until the dialog had been displayed and dealt with by the user. That being

said, most modern GUI toolkits take the approach Android does and have dialogs be non-blocking. Some developers try to figure out some way of hacking a blocking approach on top of Android's non-blocking dialogs — their time would be far better spent learning modern event-driven programming.

Advanced ListViews

The humble `ListView` is the backbone of many an Android application. On phone-sized screens, the screen may be dominated by a single `ListView`, to allow the user to choose something to examine in more detail (e.g., pick a contact). On larger screens, the `ListView` may be shown side-by-side with the details of the selected item, to minimize the "pogo stick" effect seen on phones as users bounce back and forth between the list and the details.

While we have covered the basics of `ListView` in the core chapters of this book, there is a lot more that you can do if you so choose, to make your lists that much more interesting — this chapter will cover some of these techniques.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on `Adapter` and `AdapterView`.

Multiple Row Types, and Self Inflation

When we originally looked at `ListView`, we had all of our rows come from a common layout. Hence, while the data in each row would vary, the row structure itself would be consistent for all rows. This is very easy to set up, but it is not always what you want. Sometimes, you want a mix of row structures, such as header rows versus detail rows, or detail rows that vary a bit in structure based on the data:

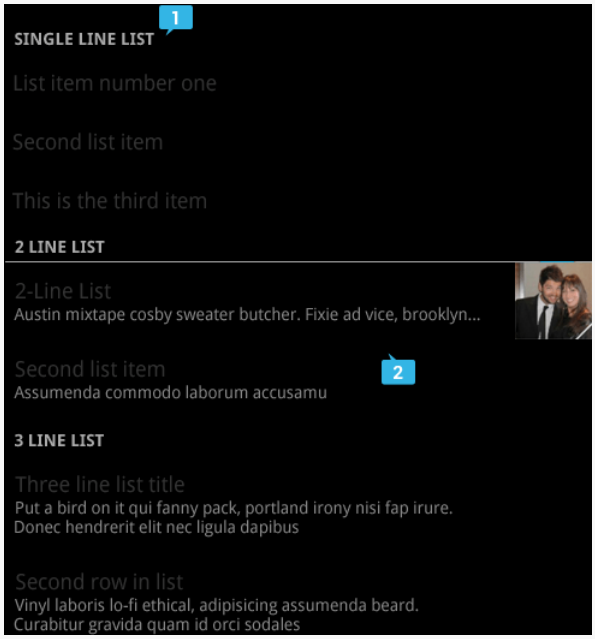


Figure 473: ListView with Row Structure Mix (image courtesy of Google)

Here, we see some header rows (e.g., "SINGLE LINE LIST") along with detail rows. While the detail rows visually vary a bit, they might still be all inflated from the same layout, simply making some pieces (second line of text, thumbnail, etc.) visible or invisible as needed. However, the header rows are sufficiently visually distinct that they really ought to come from separate layouts.

The good news is that Android supports multiple row types. However, this comes at a cost: you will need to handle the row creation yourself, rather than chaining to the superclass.

Our sample project, `Selection/HeaderDetailList` will demonstrate this, along with showing how you can create your own custom adapter straight from `BaseAdapter`, for data models that do not quite line up with what Android supports natively.

Our Data Model and Planned UI

The `HeaderDetailList` project is based on the `ViewHolderDemo` project from the chapter on `ListView`. However, this time, we have our list of 25 Latin words broken down into five groups of five, as seen in the `HeaderDetailList` activity:

```
private static final String[][] items= {
    { "lorem", "ipsum", "dolor", "sit", "amet" },
```

```
      { "consectetuer", "adipiscing", "elit", "morbi", "vel" },
      { "ligula", "vitae", "arcu", "aliquet", "mollis" },
      { "etiam", "vel", "erat", "placerat", "ante" },
      { "porttitor", "sodales", "pellentesque", "augue", "purus" } };
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

We want to display a header row for each batch:

Figure 474: HeaderDetailList, on Android 4.0.3

The Basic BaseAdapter

Once again, we have a custom `ListAdapter` named `IconicAdapter`. However, this time, instead of inheriting from `ArrayAdapter`, or even `CursorAdapter`, we are inheriting from `BaseAdapter`. As the name suggests, `BaseAdapter` is a basic implementation of the `ListAdapter` interface, with stock implementations of many of the `ListAdapter` methods. However, `BaseAdapter` is `abstract`, and so there are a few methods that we need to implement:

- `getCount()` returns the total number of rows that would be in the list. In our case, we total up the sizes of each of the batches, plus add one for each batch for our header rows:

```
@Override
public int getCount() {
  int count=0;

  for (String[] batch : items) {
    count+=1 + batch.length;
  }

  return(count);
}
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

- `getItem()` needs to return the data model for a given position, passed in as the typical `int` index. An `ArrayAdapter` would return the value out of the array at that index; a `CursorAdapter` would return the `Cursor` positioned at that row. In our case, we will return one of two objects: either the `String` for rows that are to display a Latin word, or an `Integer` containing our batch's index for rows that are to be a header:

```
@Override
public Object getItem(int position) {
  int offset=position;
  int batchIndex=0;

  for (String[] batch : items) {
    if (offset == 0) {
      return(Integer.valueOf(batchIndex));
    }

    offset--;

    if (offset < batch.length) {
      return(batch[offset]);
    }

    offset-=batch.length;
    batchIndex++;
  }

  throw new IllegalArgumentException("Invalid position: "
      + String.valueOf(position));
}
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

- `getItemId()` needs to return a unique `long` value for a given position. A `CursorAdapter` would find the `_id` value in the `Cursor` for that position and return it. In our case, lacking anything else, we simply return the position itself:

```
@Override
public long getItemId(int position) {
  return(position);
}
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

- `getView()`, which returns the `View` to use for a given row. This is the method that we overrode on our `IconicAdapter` in some previous incarnations to tailor the way the rows were populated. Our `getView()` implementation will be a bit more complex in this case, due to our multiple-row-type requirement, so we will examine it a bit later in this section.

Requesting Multiple Row Types

The methods listed above are the `abstract` ones that you have no choice but to implement yourself. Anything else on the `ListAdapter` interface that you wish to override you can, to replace the stub implementation supplied by `BaseAdapter`.

If you wish to have more than one type of row, there are two such methods that you will wish to override:

- `getViewTypeCount()` needs to return the number of distinct row types you will use. In our case, there are just two:

```
@Override
public int getViewTypeCount() {
  return(2);
}
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

- `getItemViewType()` needs to return a value from `0` to `getViewTypeCount()-1`, indicating the index of the particular row type to use for a particular row position. In our case, we need to return different values for headers (`0`) and detail rows (`1`). To determine which is which, we use `getItem()` — if we get an `Integer` back, we need to use a header row for that position:

```
  @Override
  public int getItemViewType(int position) {
    if (getItem(position) instanceof Integer) {
      return(0);
    }

    return(1);
  }
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

The reason for supplying this information is for row recycling. The `View` that is passed into `getView()` is either `null` or a row that we had previously created that has scrolled off the screen. By passing us this now-unused `View`, Android is asking us to reuse it if possible. By specifying the row type for each position, Android will ensure that it hands us the right type of row for recycling — we will not be passed in a header row to recycle when we need to be returning a detail row, for example.

Creating and Recycling the Rows

Our `getView()` implementation, then, needs to have two key enhancements over previous versions:

1. We need to create the rows ourselves, particularly using the appropriate layout for the required row type (header or detail)
2. We need to recycle the rows when they are provided, as this has a **major** impact on the scrolling speed of our `ListView`

To help simplify the logic, we will have `getView()` focus on the detail rows, with a separate `getHeaderView()` to create/recycle and populate the header rows. Our `getView()` determines up front whether the row required is a header and, if so, delegates the work to `getHeaderView()`:

```
  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    if (getItemViewType(position) == 0) {
      return(getHeaderView(position, convertView, parent));
    }

    View row=convertView;

    if (row == null) {
      row=getLayoutInflater().inflate(R.layout.row, parent, false);
    }
```

```
    ViewHolder holder=(ViewHolder)row.getTag();

    if (holder == null) {
      holder=new ViewHolder(row);
      row.setTag(holder);
    }

    String word=(String)getItem(position);

    if (word.length() > 4) {
      holder.icon.setImageResource(R.drawable.delete);
    }
    else {
      holder.icon.setImageResource(R.drawable.ok);
    }

    holder.label.setText(word);
    holder.size.setText(String.format(getString(R.string.size_template),
                                      word.length()));

    return(row);
  }
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

Assuming that we are to create a detail row, we then check to see if we were passed in a non-`null` `View`. If we were passed in `null`, we cannot recycle that row, so we have to inflate a new one via a call to `inflate()` on a `LayoutInflater` we get via `getLayoutInflater()`. But, if we were passed in an actual `View` to recycle, we can skip this step.

From here, the `getView()` implementation is largely the way it was before, including dealing with the `ViewHolder`. The only change of significance is that we have to manage the `label` `TextView` ourselves — before, we chained to the superclass and let `ArrayAdapter` handle that. So our `ViewHolder` now has a `label` data member with our `label` `TextView`, and we fill it in along with the `size` and `icon`. Also, we use `getItem()` to retrieve our Latin word, so it can find the right word for the given position out of our various word batches.

Our `getHeaderView()` does much the same thing, except it uses `getItem()` to retrieve our batch index, and we use that for constructing our header:

```
  private View getHeaderView(int position, View convertView,
                             ViewGroup parent) {
```

```
    View row=convertView;

    if (row == null) {
      row=getLayoutInflater().inflate(R.layout.header, parent, false);
    }

    Integer batchIndex=(Integer)getItem(position);
    TextView label=(TextView)row.findViewById(R.id.label);

    label.setText(String.format(getString(R.string.batch),
                                1 + batchIndex.intValue()));

    return(row);
  }
```

(from Selection/HeaderDetailList/app/src/main/java/com/commonsware/android/headerdetail/HeaderDetailListDemo.java)

Choice Modes and the Activated Style

In the chapter on large-screen strategies, we saw the EU4You sample application, and we mentioned that the `ListView` formatted its rows as "activated" to represent the current selection, when the `ListView` was side-by-side with the details.

In the chapter on styles, we saw an example of an "activated" style that referred to a device-specific color to use for an activated background. It just so happens that this is the same style that we used in EU4You.

Hence, the recipe for using activated notation for a `ListView` adjacent to details on the last-clicked-upon `ListView` row is:

- Use `CHOICE_MODE_SINGLE` (or `android:choiceMode="singleChoice"`) on the `ListView`.
- Have a style resource, in `res/values-v11/`, that references the device-specific activated background:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/activatedBackgroundIndicator</item>
  </style>
</resources>
```

- Have the same style resource also defined in `res/values` if you are supporting pre-Honeycomb devices, where you skip the parent and the

background color override, as neither of those specific values existed before API Level 11:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated">
  </style>
</resources>
```

- Use that style as the background of your `ListView` row (e.g., `style="@style/activated"`)

Android will automatically color the row background based upon the last row clicked, instead of checking a `RadioButton` as you might ordinarily see with `CHOICE_MODE_SINGLE` lists.

Custom Mutable Row Contents

Lists with pretty icons next to them are all fine and well. But, can we create `ListView` widgets whose rows contain interactive child widgets instead of just passive widgets like `TextView` and `ImageView`? For example, there is a `RatingBar` widget that allows users to assign a rating by clicking on a set of star icons. Could we combine the `RatingBar` with text in order to allow people to scroll a list of, say, songs and rate them right inside the list?

There is good news and bad news.

The good news is that interactive widgets in rows work just fine. The bad news is that it is a little tricky, specifically when it comes to taking action when the interactive widget's state changes (e.g., a value is typed into a field). We need to store that state somewhere, since our `RatingBar` widget will be recycled when the `ListView` is scrolled. We need to be able to set the `RatingBar` state based upon the actual word we are viewing as the `RatingBar` is recycled, and we need to save the state when it changes so it can be restored when this particular row is scrolled back into view.

What makes this interesting is that, by default, the `RatingBar` has absolutely no idea what item in the `ArrayAdapter` it represents. After all, the `RatingBar` is just a widget, used in a row of a `ListView`. We need to teach the rows which item in the `ArrayAdapter` they are currently displaying, so when their `RatingBar` is checked, they know which item's state to modify.

So, let's see how this is done, using the activity in the Selection/RateList sample project. We will use the same basic classes as in most of our ListView samples, where we are showing a list of Latin words. In this case, you can rate the words on a three-star rating. Words given a top rating are put in all caps:

```
package com.commonsware.android.ratelist;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.LinearLayout;
import android.widget.RatingBar;
import android.widget.TextView;
import java.util.ArrayList;

public class RateListDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    ArrayList<RowModel> list=new ArrayList<RowModel>();

    for (String s : items) {
      list.add(new RowModel(s));
    }

    setListAdapter(new RatingAdapter(list));
  }

  private RowModel getModel(int position) {
    return(((RatingAdapter)getListAdapter()).getItem(position));
  }

  class RatingAdapter extends ArrayAdapter<RowModel> {
    RatingAdapter(ArrayList<RowModel> list) {
      super(RateListDemo.this, R.layout.row, R.id.label, list);
    }
```

```
  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    RatingBar bar=(RatingBar)row.getTag();

    if (bar==null) {
      bar=(RatingBar)row.findViewById(R.id.rate);
      row.setTag(bar);

      RatingBar.OnRatingBarChangeListener l=
                  new RatingBar.OnRatingBarChangeListener() {
        public void onRatingChanged(RatingBar ratingBar,
                                    float rating,
                                    boolean fromTouch)  {
          Integer myPosition=(Integer)ratingBar.getTag();
          RowModel model=getModel(myPosition);

          model.rating=rating;

          LinearLayout parent=(LinearLayout)ratingBar.getParent();
          TextView label=(TextView)parent.findViewById(R.id.label);

          label.setText(model.toString());
        }
      };

      bar.setOnRatingBarChangeListener(l);
    }

    RowModel model=getModel(position);

    bar.setTag(Integer.valueOf(position));
    bar.setRating(model.rating);

    return(row);
  }
}

class RowModel {
  String label;
  float rating=2.0f;

  RowModel(String label) {
    this.label=label;
  }

  public String toString() {
    if (rating>=3.0) {
```

```
        return(label.toUpperCase());
      }

      return(label);
    }
  }
}
```

(from Selection/RateList/app/src/main/java/com/commonsware/android/ratelist/RateListDemo.java)

Here is what is different in this activity and `getView()` implementation than in earlier, simpler samples:

1. While we are still using `String` array items as the list of Latin words, rather than pour that String array straight into an `ArrayAdapter`, we turn it into a list of `RowModel` objects. `RowModel` is the mutable model: it holds the Latin word plus the current rating. In a real system, these might be objects populated from a database, and the properties would have more business meaning.
2. Utility methods like `onListItemClick()` had to be updated to reflect the change from a pure-String model to use a `RowModel`.
3. The `ArrayAdapter` subclass (`RatingAdapter`), in `getView()`, lets `ArrayAdapter` inflate and recycle the row, then checks to see if we have a `ViewHolder` in the row's tag. If not, we create a new `ViewHolder` and associate it with the row. For the row's `RatingBar`, we add an anonymous `onRatingChanged()` listener that looks at the row's tag (`getTag()`) and converts that into an `Integer`, representing the position within the `ArrayAdapter` that this row is displaying. Using that, the rating bar can get the actual `RowModel` for the row and update the model based upon the new state of the rating bar. It also updates the text adjacent to the `RatingBar` when checked to match the rating bar state.
4. We always make sure that the `RatingBar` has the proper contents and has a tag (via `setTag()`) pointing to the position in the adapter the row is displaying.

The row layout is very simple: just a `RatingBar` and a `TextView` inside a `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
```

```
>
  <RatingBar
    android:id="@+id/rate"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="3"
    android:stepSize="1"
    android:rating="2" />
  <TextView
    android:id="@+id/label"
    android:padding="2dip"
    android:textSize="18sp"
    android:layout_gravity="left|center_vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
</LinearLayout>
```

(from Selection/RateList/app/src/main/res/layout/row.xml)

And the result is what you would expect, visually:

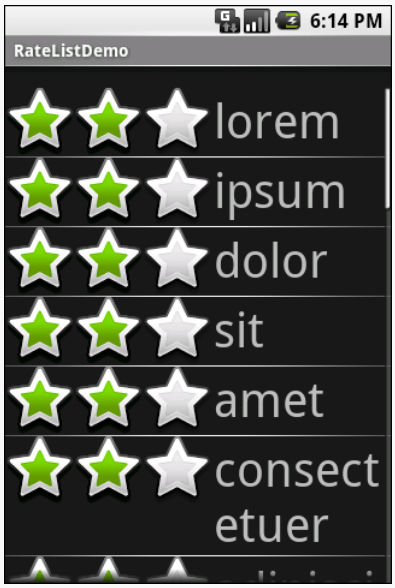


Figure 475: RateList, As Initially Shown

This includes the toggled rating bars turning their words into all caps:

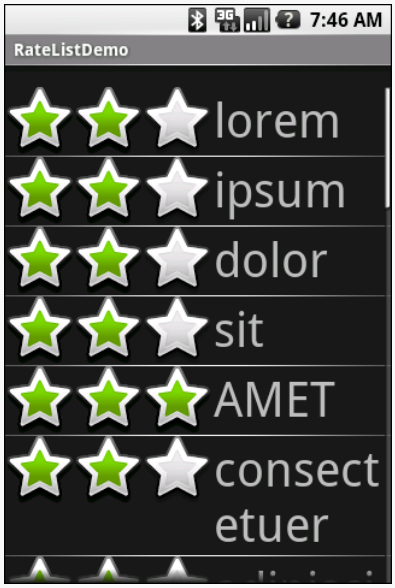


Figure 476: RateList, With a Three-Star Word

From Head To Toe

Perhaps you do not need section headers scattered throughout your list. If you only need extra "fake rows" at the beginning or end of your list, you can use header and footer views.

`ListView` supports `addHeaderView()` and `addFooterView()` methods that allow you to add `View` objects to the beginning and end of the list, respectively. These `View` objects otherwise behave like regular rows, in that they are part of the scrolled area and will scroll off the screen if the list is long enough. If you want fixed headers or footers, rather than put them in the `ListView` itself, put them outside the `ListView`, perhaps using a `LinearLayout`.

To demonstrate header and footer views, take a peek at the `Selection/HeaderFooter` sample project, particularly the `HeaderFooterDemo` class:

```
package com.commonsware.android.header;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import android.app.ListActivity;
import android.os.Bundle;
import android.os.SystemClock;
```

```
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.TextView;

public class HeaderFooterDemo extends ListActivity {
  private static String[] items={"lorem", "ipsum", "dolor",
                                 "sit", "amet", "consectetuer",
                                 "adipiscing", "elit", "morbi",
                                 "vel", "ligula", "vitae",
                                 "arcu", "aliquet", "mollis",
                                 "etiam", "vel", "erat",
                                 "placerat", "ante",
                                 "porttitor", "sodales",
                                 "pellentesque", "augue",
                                 "purus"};
  private long startTime=SystemClock.uptimeMillis();
  private boolean areWeDeadYet=false;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
    getListView().addHeaderView(buildHeader());
    getListView().addFooterView(buildFooter());
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        items));
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    areWeDeadYet=true;
  }

  private View buildHeader() {
    Button btn=new Button(this);

    btn.setText("Randomize!");
    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        List<String> list=Arrays.asList(items);

        Collections.shuffle(list);

        setListAdapter(new ArrayAdapter<String>(HeaderFooterDemo.this,
```

```
                            android.R.layout.simple_list_item_1,
                            list));
      }
    });

    return(btn);
  }

  private View buildFooter() {
    TextView txt=new TextView(this);

    updateFooter(txt);

    return(txt);
  }

  private void updateFooter(final TextView txt) {
    long runtime=(SystemClock.uptimeMillis()-startTime)/1000;

    txt.setText(String.valueOf(runtime)+" seconds since activity launched");

    if (!areWeDeadYet) {
      getListView().postDelayed(new Runnable() {
        public void run() {
          updateFooter(txt);
        }
      }, 1000);
    }
  }
}
```

(from Selection/HeaderFooter/app/src/main/java/com/commonsware/android/header/HeaderFooterDemo.java)

Here, we add a header `View` built via `buildHeader()`, returning a `Button` that, when clicked, will shuffle the contents of the list. We also add a footer `View` built via `buildFooter()`, returning a `TextView` that shows how long the activity has been running, updated every second. The list itself is the ever-popular list of *lorem ipsum* words.

When initially displayed, the header is visible but the footer is not, because the list is too long:

Figure 477: A ListView with a header view shown

If you scroll downward, the header will slide off the top, and eventually the footer will scroll into view:

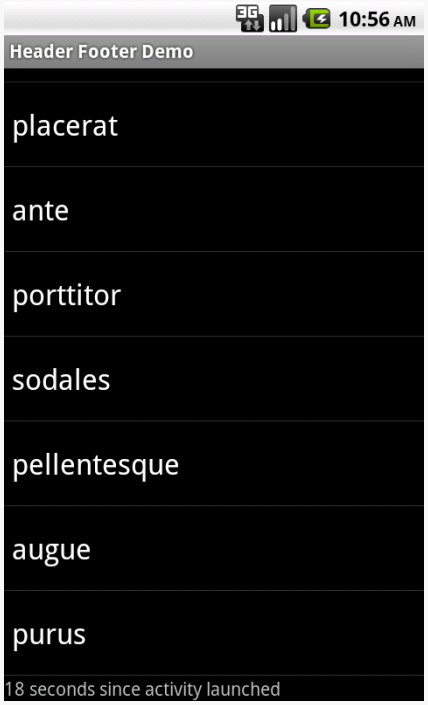


Figure 478: A ListView with a footer view shown

Enter RecyclerView

`RecyclerView` is a more powerful (and more complex) replacement for `ListView` and `GridView`. You can read more about what it does and how you can use it.

If you have spent much time on an Android 3.0+ device, then you probably have run into a curious phenomenon. Sometimes, when you select an item in a list or other widget, the action bar magically transforms from its normal look:

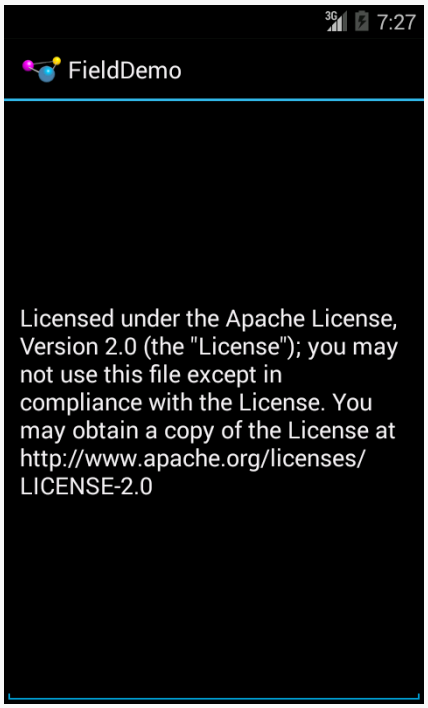


Figure 479: Regular Action Bar for Activity with EditText

to one designed to perform operations on what you have selected:

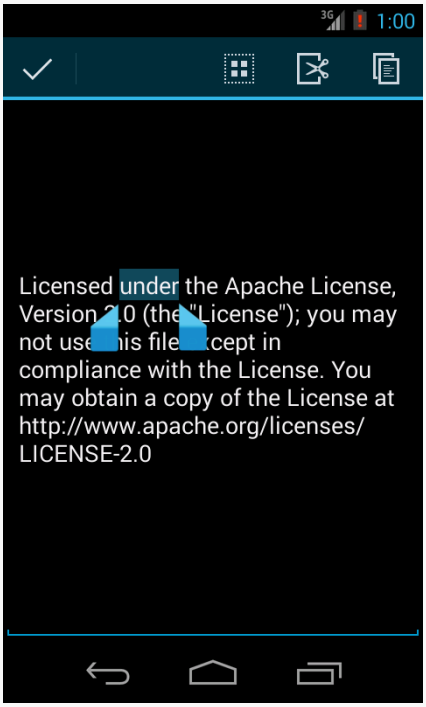


Figure 480: Action Mode, Given Selected Word in EditText

The good news is that this is not some sort of magic limited only to built-in widgets like `EditText`. You too can have this effect in your application, by triggering an "action mode".

In this chapter, we will explore how you can set up and respond to action modes.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

A Matter of Context

Most desktop operating systems have had the notion of a "context menu" for some time, typically triggered by a click of the right mouse button. In particular, a right-click over some selected item might bring up a context menu of operations to perform on that item:

- Selecting text in a text editor, then right-clicking, might bring up a context menu for cut/copy/paste of the text
- Right-clicking over a file in some sort of file explorer might bring up a context menu for cut/copy/paste of the file
- Etc.

Android supports context menus, driven by a long-tap on a widget rather than a right-click. You will find a few applications that offer such menus, particularly on lists of things. However, context menus are a very old UI design pattern in Android, and modern apps rarely use them.

Instead, contextual operations are raised via an action mode, so when the user specifies a context (e.g., selects a word in an `EditText`), the action bar changes to show operations relevant for the selection.

Manual Action Modes

A common pattern will be to activate an action mode when the user checks off something in a multiple-choice `ListView`. If you want to go that route, there is some built-in scaffolding to make that work, described later in this chapter.

You can, if you wish, move the action bar into an action mode whenever you want. This would be particularly important if your UI is not based on a `ListView`. For example, tapping on an image in a `GridView` might activate it and move you into an action mode for operations upon that particular image.

In this section, we will examine the `ActionMode/ManualNative` sample project. This is another variation on the "show a list of Latin words in a list" sample used elsewhere in this book.

Choosing Your Trigger

As mentioned above, selecting a word or passage in an `EditText` (e.g., via a long-tap) brings up an action mode for cut/copy/paste operations. Other apps might bring up an action mode when you check an item in a checklist. Yet others might bring up an action mode when you long-tap on an item in a regular list. And so on.

You will need to choose, for your own UI, what trigger mechanism will bring up an action mode. It should be some trigger that makes it obvious to the user what the action mode will be acting upon. For example:

- If the user long-taps on an item in a `GridView`, bring up an action mode, and treat future taps on `GridView` items as adding or removing items from the "selection" while that action mode is visible
- If the user "rubber-bands" some figures in your vector art drawing `View`, bring up an action mode for operations on those figures (e.g., rotate, resize)
- And so on

In the case of the sample project, we stick with the classic long-tap on a `ListView` row to bring up an action mode:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  initAdapter();
  getListView().setLongClickable(true);
  getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
  getListView().setOnItemLongClickListener(new ActionModeHelper(
                                                                this,
                                                                getListView()));
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeDemo.java)

Starting the Action Mode

Starting an action mode is trivially easy: just call `startActionMode()` on your `Activity`, passing in an implementation of `ActionMode.Callback`, which will be called with various lifecycle methods for the action mode itself.

In the case of the `ActionMode` sample project, `ActionModeHelper` – our `OnItemLongClickListener` from the preceding section – also is our `ActionMode.Callback` implementation. Hence, when the user long-clicks on an item in the `ListView`, the `ActionModeHelper` establishes itself as the action mode:

```
@Override
public boolean onItemLongClick(AdapterView<?> view, View row,
                               int position, long id) {
  modeView.clearChoices();
  modeView.setItemChecked(position, true);

  if (activeMode == null) {
    activeMode=host.startActionMode(this);
  }

  return(true);
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeHelper.java)

Note that `startActionMode()` returns an `ActionMode` object, which we can use later on to configure the mode's behavior, by stashing it in an `actionMode` data member.

Also, we make the long-clicked-upon item be "checked", to show which item the action mode will act upon. Our row layout will make a checked row show up with the "activated" style, courtesy of Android's `simple_list_item_activated_1` stock layout.

Also note that we only start the action mode if it is not already started.

Implementing the Action Mode

The real logic behind the action mode lies in your `ActionMode.Callback` implementation. It is in these four lifecycle methods where you define what the action mode should look like and what should happen when choices are made in it.

onCreateActionMode()

The `onCreateActionMode()` method will be called shortly after you call `startActionMode()`. Here, you get to define what goes in the action mode. You get the `ActionMode` object itself (in case you do not already have a reference to it). More importantly, you are passed a `Menu` object, just as you get in `onCreateOptionsMenu()`. And, just like with `onCreateOptionsMenu()`, you can inflate a menu resource into the `Menu` object to define the contents of the action mode:

```
@Override
public boolean onCreateActionMode(ActionMode mode, Menu menu) {
  MenuInflater inflater=host.getMenuInflater();

  inflater.inflate(R.menu.context, menu);
  mode.setTitle(R.string.context_title);

  return(true);
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeHelper.java)

In addition to inflating our menu resource into the action mode's menu, we also set the title of the `ActionMode`, which shows up to the right of the Done button:

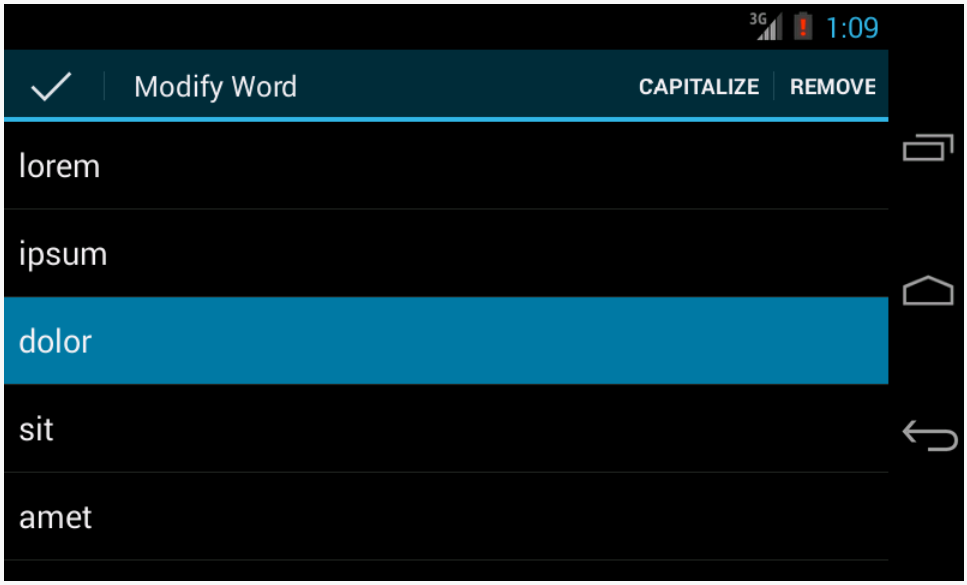


Figure 481: The ManualNative Sample App, Showing an Action Mode

onPrepareActionMode()

If you determine that you need to change the contents of your action mode, you can call `invalidate()` on the `ActionMode` object. That, in turn, will trigger a call to `onPrepareActionMode()`, where you once again have an opportunity to configure the `Menu` object. If you do make changes, return `true` — otherwise, return `false`. In the case of `ActionModeHelper`, we take the latter approach:

```
@Override
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
  return(false);
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeHelper.java)

onActionItemClicked()

Just as `onCreateActionMode()` is the action mode analogue to `onCreateOptionsMenu()`, `onActionItemClicked()` is the action mode analogue to `onOptionsItemSelected()`. This will be called if the user clicks on something related to your action mode. You are passed in the corresponding `MenuItem` object (plus the `ActionMode` itself), and you can take whatever steps are necessary to do whatever the work is.

On the `ActionModeDemo` class, we have the business logic for handling the data-change operations in a `performAction()` method:

```
public boolean performAction(int itemId, int position) {
  switch (itemId) {
    case R.id.cap:
      String word=words.get(position);

      word=word.toUpperCase();

      adapter.remove(words.get(position));
      adapter.insert(word, position);

      return(true);

    case R.id.remove:
      adapter.remove(words.get(position));

      return(true);
  }

  return(false);
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeDemo.java)

And, the `onActionItemClicked()` method calls `performAction()`:

```
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
  boolean result=
      host.performAction(item.getItemId(),
                         modeView.getCheckedItemPosition());

  if (item.getItemId() == R.id.remove) {
    activeMode.finish();
  }

  return(result);
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeHelper.java)

`onActionItemClicked()` also dismisses the action mode if the user chose the "remove" item, since the action mode is no longer needed. You get rid of an active action mode by calling `finish()` on it.

onDestroyActionMode()

The `onDestroyActionMode()` callback will be invoked when the action mode goes away, for any reason, such as:

1. The user clicks the Done button on the left
2. The user clicks the BACK button
3. You call `finish()` on the `ActionMode`

Here, you can do any necessary cleanup. `ActionModeHelper` tries to clean things up, notably the "checked" state of the last item long-tapped-upon:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  activeMode=null;
  modeView.clearChoices();
  modeView.requestLayout();
}
```

(from ActionMode/ManualNative/app/src/main/java/com/commonsware/android/actionmode/ActionModeHelper.java)

However, for reasons that are not yet clear, `clearChoices()` does not update the UI when called from `onDestroyActionMode()` unless you also call `requestLayout()`.

Multiple-Choice-Modal Action Modes

For many cases, the best user experience will be for you to have a multiple-choice `ListView`, where checking items in that list enables an action mode for performing operations on the checked items. For this scenario, API Level 11+ has a built-in `ListView` choice mode, `CHOICE_MODE_MULTIPLE_MODAL`, that automatically sets up an `ActionMode` for you as the user checks and unchecks items.

To see how this works, let's examine the `ActionMode/ActionModeMC` sample project. This is the same project as in the preceding section, but altered to have a multiple-choice `ListView`, utilizing an action mode on API Level 11+.

Once again, in `onCreate()`, we need to set up the smarts for our `ListView`. This time, though, we will use `CHOICE_MODE_MULTIPLE_MODAL`:

```
@Override
 public void onCreate(Bundle state) {
   super.onCreate(state);
```

```
    initAdapter();

    getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
    getListView().setMultiChoiceModeListener(new HCMultiChoiceModeListener(
        this, getListView()));
  }
```

(from ActionMode/ActionModeMC/app/src/main/java/com/commonsware/android/actionmodemc/ActionModeDemo.java)

We enable `CHOICE_MODE_MULTIPLE_MODAL` for the `ListView`, and register an instance of an `HCMultiChoiceModeListener` object via `setMultiChoiceModeListener()`. This object is an implementation of the `MultiChoiceModeListener` interface that we will examine shortly.

Since we now may have multiple checked items, our `performAction()` method must take this into account, capitalizing or removing all checked words:

```
  public boolean performActions(MenuItem item) {
    SparseBooleanArray checked=getListView().getCheckedItemPositions();

    switch (item.getItemId()) {
      case R.id.cap:
        for (int i=0; i < checked.size(); i++) {
          if (checked.valueAt(i)) {
            int position=checked.keyAt(i);
            String word=words.get(position);

            word=word.toUpperCase(Locale.ENGLISH);

            adapter.remove(words.get(position));
            adapter.insert(word, position);
          }
        }

        return(true);

      case R.id.remove:
        ArrayList<Integer> positions=new ArrayList<Integer>();

        for (int i=0; i < checked.size(); i++) {
          if (checked.valueAt(i)) {
            positions.add(checked.keyAt(i));
          }
        }

        Collections.sort(positions, Collections.reverseOrder());
```

```
      for (int position : positions) {
        adapter.remove(words.get(position));
      }

      getListView().clearChoices();

      return(true);
    }

    return(false);
  }
```

(from ActionMode/ActionModeMC/app/src/main/java/com/commonsware/android/actionmodemc/ActionModeDemo.java)

`MultiChoiceModeListener` extends the `ActionMode.Callback` interface we used with our manual action mode earlier in this book. Hence, we need to implement all the standard `ActionMode.Callback` methods, plus a new `onItemCheckedStateChanged()` method introduced by `MultiChoiceModeListener`:

```
package com.commonsware.android.actionmodemc;

import android.annotation.TargetApi;
import android.os.Build;
import android.view.ActionMode;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.AbsListView;
import android.widget.ListView;

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public class HCMultiChoiceModeListener implements
    AbsListView.MultiChoiceModeListener {
  ActionModeDemo host;
  ActionMode activeMode;
  ListView lv;

  HCMultiChoiceModeListener(ActionModeDemo host, ListView lv) {
    this.host=host;
    this.lv=lv;
  }

  @Override
  public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    MenuInflater inflater=host.getMenuInflater();
```

```
    inflater.inflate(R.menu.context, menu);
    mode.setTitle(R.string.context_title);
    mode.setSubtitle("(1)");
    activeMode=mode;

    return(true);
  }

  @Override
  public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
    return(false);
  }

  @Override
  public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    boolean result=host.performActions(item);

    updateSubtitle(activeMode);

    return(result);
  }

  @Override
  public void onDestroyActionMode(ActionMode mode) {
    activeMode=null;
  }

  @Override
  public void onItemCheckedStateChanged(ActionMode mode, int position,
                                        long id, boolean checked) {
    updateSubtitle(mode);
  }

  private void updateSubtitle(ActionMode mode) {
    mode.setSubtitle("(" + lv.getCheckedItemCount() + ")");
  }
}
```

(from ActionMode/ActionModeMC/app/src/main/java/com/commonsware/android/actionmodemc/HCMultiChoiceModeListener.java)

Android will automatically start our action mode for us when the user checks the first item in the list, using our `MultiChoiceModeListener` as the callback. Android will also automatically finish the action mode if the user unchecks all previously-checked items.

In `onCreateActionMode()`, we populate the menu, plus set up a title and subtitle on the `ActionMode`. The subtitle appears below the title, as you might expect. In this

case, we are indicating how many words are checked and therefore will be affected by the actions the user chooses in the action mode:

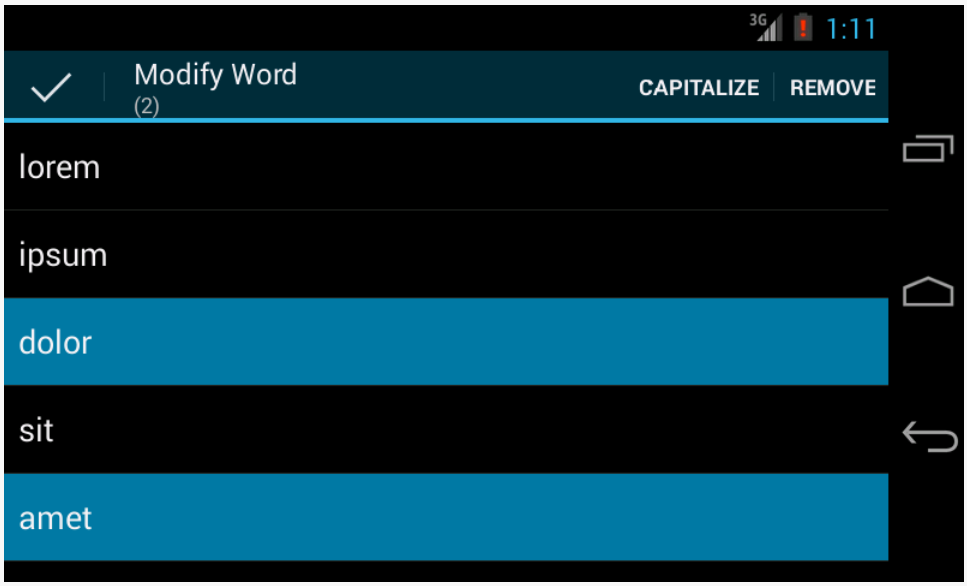


Figure 482: The ActionModeMC Sample App, Showing the Action Mode

Then, in `onActionItemClicked()`, we both call `performActions()` to affect the desired changes, plus update the subtitle in case the user removed words (which means they are no longer checked).

The new `onItemCheckedStateChanged()` will be called whenever the user checks or unchecks an item, up until the last item is unchecked. `HCMultiChoiceModeListener` simply updates the subtitle to reflect the new count of checked items.

On the whole, using `CHOICE_MODE_MULTIPLE_MODAL` is simpler than setting up your own trigger mechanism and managing the action mode yourself. That being said, both are completely valid options, which is particularly important for situations where a multiple-choice `ListView` is not the desired user interface.

Long-Click To Initiate an Action Mode

However, rather than having checkboxes or the like always in the `ListView`, a more modern approach is to move into multiple-selection mode based on a long-click. Before then, clicks on rows behave like with any other `ListView`, but after a long-

click, the action mode appears and the user can tap on rows to select which of them to operate upon.

The ActionMode/LongPress sample project is a variation on the preceding project, with some slight simplifications, and adopting the long-click as the means to enter the action mode.

Setting Up the Listeners

In `onCreate()`, we set up listeners for both a long click (via `setOnItemLongClickListener()`) and for multiple-choice mode (via `setMultiChoiceModeListener()`. Both times, we supply the activity as the listener, as it implements the appropriate interfaces:

```
    getListView().setOnItemLongClickListener(this);
    getListView().setMultiChoiceModeListener(this);
```

(from ActionMode/LongPress/app/src/main/java/com/commonsware/android/actionmode/longpress/ActionModeDemo.java)

Handling the Long Click

By default, the `ListView` is in no-choice mode, where clicks on rows simply trigger `onListItemClick()` or the equivalent. However, if the user long-clicks on a row, our `onItemLongClick()` method will be called, and we can both switch into multiple-choice mode and mark the long-clicked row as being checked:

```
@Override
public boolean onItemLongClick(AdapterView<?> parent, View view,
                               int position, long id) {
  getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
  getListView().setItemChecked(position, true);

  return(true);
}
```

(from ActionMode/LongPress/app/src/main/java/com/commonsware/android/actionmode/longpress/ActionModeDemo.java)

At this point, the action mode will also start up, courtesy of having called `setMultiChoiceModeListener()`.

Addressing Configuration Changes

If we undergo a configuration change, we want:

1. To keep the current set of words, including any that were added
2. To keep the action mode going, if the user had long-clicked to enter the action mode
3. To keep our checked item states, if the action mode is active

Keeping the checked item states will be handled for us by the built-in instance-state management of `ListView` and `ListActivity`. However, the rest we need to handle ourselves. So, we have an `onSaveInstanceState()` implementation in the activity, which saves the current choice mode, plus the current word list:

```
@Override
public void onSaveInstanceState(Bundle state) {
  super.onSaveInstanceState(state);
  state.putInt(STATE_CHOICE_MODE, getListView().getChoiceMode());
  state.putStringArrayList(STATE_MODEL, words);
}
```

(from ActionMode/LongPress/app/src/main/java/com/commonsware/android/actionmode/longpress/ActionModeDemo.java)

Plus, in `onCreate()`, after setting up the listeners, we set up the choice mode of the `ListView` based upon the passed in instance state `Bundle`, if there is one:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  if (state == null) {
    initAdapter(null);
  }
  else {
    initAdapter(state.getStringArrayList(STATE_MODEL));
  }

  getListView().setOnItemLongClickListener(this);
  getListView().setMultiChoiceModeListener(this);

  int choiceMode=
      (state == null ? ListView.CHOICE_MODE_NONE
          : state.getInt(STATE_CHOICE_MODE));

  getListView().setChoiceMode(choiceMode);
}
```

(from ActionMode/LongPress/app/src/main/java/com/commonsware/android/actionmode/longpress/ActionModeDemo.java)

Once we call `setChoiceMode()` with the previous activity instance's choice mode, if that was `CHOICE_MODE_MULTIPLE_MODAL`, Android will automatically open up the action mode again and restore our checked items.

Resetting the Choice Mode

Where things get a bit interesting is when the user dismisses the action mode, at which point we need to move back to no-choice mode.

You might think that this would merely be a matter of calling `setChoiceMode()` on the `ListView`, asking for `CHOICE_MODE_NONE`. Indeed, that is part of the solution. However, there are two problems:

1. If you call that in `onDestroyActionMode()` directly, you wind up with infinite recursion and a `StackOverflowError`, as changing the choice mode while the action mode is still technically active will cause it to destroy the action mode again.
2. Switching the choice mode back to "none" enables some optimizations within `ListView` that ignore the checked state of our rows. However, those rows still already checked will show up as activated, even after calling `setChoiceMode()` to return to the normal "none" mode. `clearChoices()` also does not have a worthwhile effect, for whatever reason.

Hence, in `onDestroyActionMode()`, not only do we need to call `setChoiceMode()`, but we need to "smack around" the `ListView` enough to get it to clear our checked rows, and the easiest way to do that is to call `setAdapter()` on it, passing in its existing adapter:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  if (activeMode != null) {
    activeMode=null;
    getListView().setChoiceMode(ListView.CHOICE_MODE_NONE);
    getListView().setAdapter(getListView().getAdapter());
  }
}
```

(from ActionMode/LongPress/app/src/main/java/com/commonsware/android/actionmode/longpress/ActionModeDemo.java)

And, we only do that while our action mode is active (i.e., `activeMode` is not `null`), to avoid the infinite recursion.

This is a bit clunky, but it works.

The Results

When initially launched, the activity looks like a simple `ListActivity`:

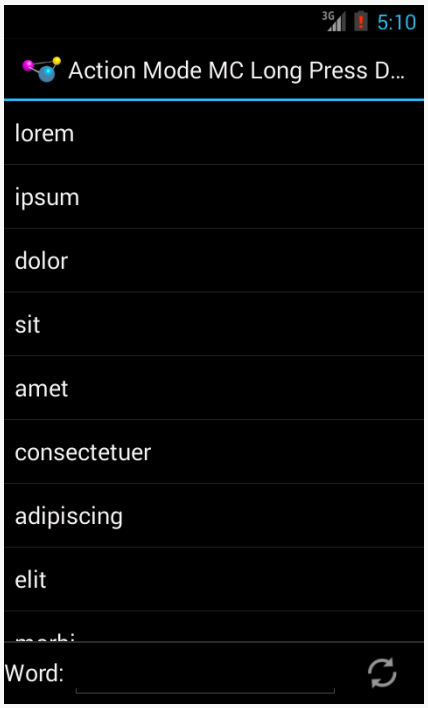


Figure 483: Action Mode Long Press Demo, As Initially Launched

Tapping on a row provides the normal momentary highlight.

However, if the user long-clicks a row, we move into the action mode and a multiple-choice `ListView`:

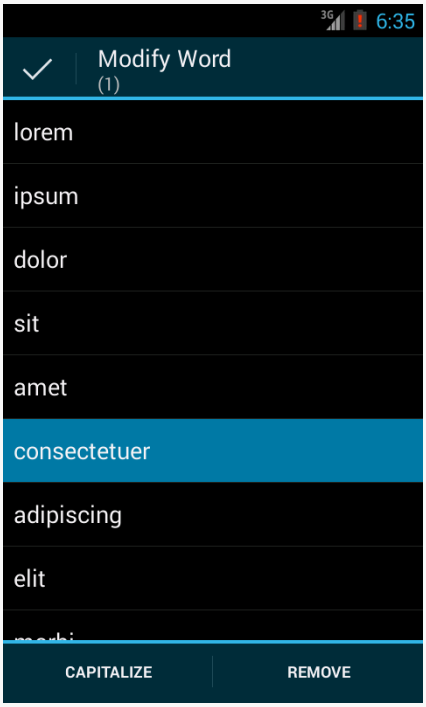


Figure 484: Action Mode Long Press Demo, with Action Mode Activated

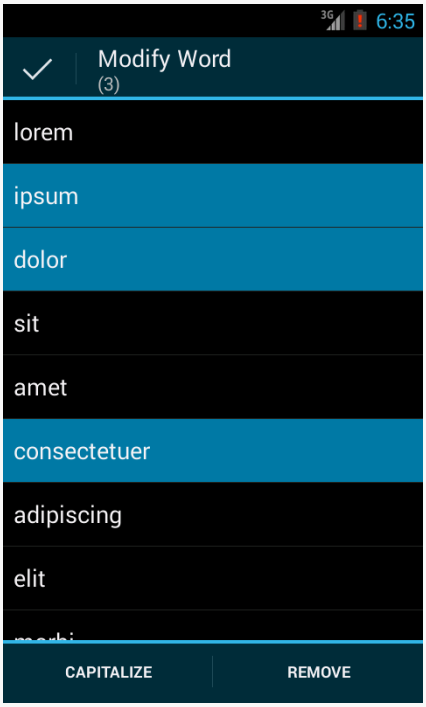


Figure 485: Action Mode Long Press Demo, with Multiple Selections

Dismissing the action mode returns the `ListView` to normal operation.

Other Advanced Action Bar Techniques

The action bar offers a number of other features that developers can take advantage of, ones that do not necessarily fit into the other chapters. Hence, this chapter is a "catch all" for other things you may wish to do with your action bar. Note that this chapter is focused on the native action bar, not the AppCompat backport.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Action Layouts

What happens if you want something other than a button to appear as an action bar item? Suppose you want a field instead?

Fortunately, this is supported. Otherwise, this would be a completely pointless section of the book.

You can specify `android:actionLayout` on an `<item>` element in a menu resource. This will point to a reference to a layout XML resource that you want to have inflated into the action bar instead of a toolbar button. Then, in `onCreateOptionsMenu()`, you can call `findMenuItem()` on the `Menu` to retrieve the `MenuItem` associated with this `<item>` element, then call `getActionView()` to retrieve the root of your inflated layout. At that point, you can hook up event listeners to the widgets in that layout, as needed.

Obviously, since the action bar is only so big, you will need to be judicious about your use of space.

Action Views and Action Providers

If all you need is a single widget to replace the toolbar button, rather than a whole layout resource, you can use `android:actionViewClass` instead of `android:actionLayout`. In `android:actionViewClass`, you provide the fully-qualified class name of the widget that you wish to use to replace the toolbar button. You still use `getActionView()` to retrieve a reference to this at runtime.

If the widget you use implements the `CollapsibleActionView` interface, then it has an additional behavior: the ability to collapse into a standard toolbar button or expand into its normal mode. The only example of this in the current Android SDK is `SearchView`, which can expand into a field for searching or collapse into a simple search icon (magnifying glass) as needed. We will see more about `SearchView`, and how it behaves as a `CollapsibleActionView`, later in this chapter.

Yet another possible toolbar button replacement is an action provider. Whereas an action view or action layout provide the UI, and your code provides the handling of touch events, an action provider is an "all-in-one" solution. It is designed to be configured, then used by the user without any required additional intervention by the developer. That being said, an action provider can have its own listener interfaces to let developers know about various events that have occurred. The two primary implementations of the `ActionProvider` base class are:

- `MediaRouteActionProvider`, covered later elsewhere in the book, is used to allow users to control the destination for media, such as routing audio to Bluetooth headphones instead of the device speaker or playing content back on a Chromecast
- `ShareActionProvider` can simplify sharing content via `ACTION_SEND`, as is covered elsewhere in the book

To use an `ActionProvider`, you add the `android:actionProviderClass` attribute to an `<item>` in the `<menu>` resource, providing the fully-qualified class name of the `ActionProvider` implementation. You can call `getActionProvider()` on the `MenuItem` to retrieve the `ActionProvider` instance, for configuration at runtime.

Searching with SearchView

Many apps employ a `SearchView` in their action bar. The user typically sees the search icon as a regular toolbar button:

Figure 486: SearchView Demo, Showing Collapsed Action View

Tapping that opens a search field, taking over more of the action bar:

Figure 487: SearchView Demo, Showing Expanded Action View

Typing something in initiates some sort of search, as defined by the activity that is using the `SearchView`. BACK or the app icon in the action bar will "collapse" the `SearchView` back into its iconified state.

The `ActionBar/SearchView` sample project, profiled in this section, shows how you can use `SearchView` within your app. This sample is a clone of one of the previous action bar samples, where we have the list of 25 words, hosted in a `ListFragment`, with action bar items to add a word and reset the word list. In this section, we will augment the sample with a `SearchView` and a filtered `ListView`.

SearchView... in the Menu Resource

The project's menu resource (`res/menu/actions.xml`) contains a regular action item (`reset`), an action item employing an action layout (`add`), and an action item containing our `SearchView` (`search`):

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/search"
    android:actionViewClass="android.widget.SearchView"
    android:icon="@drawable/ic_action_search"
    android:showAsAction="ifRoom|collapseActionView"
    android:title="@string/filter">
```

```
  </item>

</menu>
```

(from ActionBar/SearchView/app/src/main/res/menu/actions.xml)

Note that the `search` item not only has `android:actionViewClass="android.widget.SearchView"` to tie in our action view, but it also has `android:showAsAction="ifRoom|collapseActionView"`, to indicate that this action view should support collapsing and expanding.

SearchView… in the Action Bar Configuration

In `onCreateOptionsMenu()` of our `ActionBarFragment`, in addition to inflating the menu resource and calling a `configureActionItem()` method to configure the `add` action layout, we now also call a `configureSearchView()` method to configure the `SearchView`:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.actions, menu);

    configureSearchView(menu);

    super.onCreateOptionsMenu(menu, inflater);
  }
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

In `configureSearchView()`, surprisingly enough, we configure the `SearchView`:

```
  private void configureSearchView(Menu menu) {
    MenuItem search=menu.findItem(R.id.search);

    sv=(SearchView)search.getActionView();
    sv.setOnQueryTextListener(this);
    sv.setOnCloseListener(this);
    sv.setSubmitButtonEnabled(false);
    sv.setIconifiedByDefault(true);

    if (initialQuery != null) {
      sv.setIconified(false);
      search.expandActionView();
      sv.setQuery(initialQuery, true);
    }
  }
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

Specifically, we:

- Register our fragment as the `QueryTextListener` and the `OnCloseListener`, which will be covered in greater detail later in this chapter
- Disable the submit button, as we will be using the `SearchView` for filtering rather than querying
- Indicate that the `SearchView` should be collapsed ("iconified") as the default state

Also, our fragment has an `initialQuery` data member, and if that is not `null`, we expand the `SearchView` and fill in `initialQuery` as the query to be shown in the `SearchView`, also submitting it.

`initialQuery` comes from our configuration change logic, as if the user fills in something in the `SearchView` in one configuration (e.g., portrait), we do not want to lose it on a configuration change (e.g., to landscape). In our `onSaveInstanceState()` method, we save both the query from the `SearchView` and the words currently in our list:

```
@Override
public void onSaveInstanceState(Bundle state) {
  super.onSaveInstanceState(state);

  if (!sv.isIconified()) {
    state.putCharSequence(STATE_QUERY, sv.getQuery());
  }

  state.putStringArrayList(STATE_MODEL, words);
}
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

In `onActivityCreated()`, we use the `savedInstanceState` `Bundle` to populate the adapter with the previous set of words, plus store the old `SearchView`'s query in `initialQuery`:

```
@Override
public void onActivityCreated(Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);

  if (savedInstanceState == null) {
    initAdapter(null);
```

```
    }
    else {
      initAdapter(savedInstanceState.getStringArrayList(STATE_MODEL));
      initialQuery=savedInstanceState.getCharSequence(STATE_QUERY);
    }

    setHasOptionsMenu(true);
  }
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

Hence, on a configuration change, by the time `configureSearchView()` is called, we will have our `initialQuery`, if there is one, and we can set up the UI to be the same as it was in the old configuration.

SearchView… And Filtering a ListView

The `ActionBarFragment` implements the `SearchView.OnQueryTextListener` and `SearchView.OnCloseListener` interfaces, which is why we can pass `this` to `setOnQueryTextListener()` and `setOnCloseListener()` in `configureSearchView()`.

Those two interfaces require a total of three methods, described below.

onQueryTextChange()

The `onQueryTextChange()` method — required by `SearchView.OnQueryTextListener` – will be called whenever the user has changed the contents of the expanded `SearchView`, such as by typing a character. This is used when you want to employ the `SearchView` for filtering, updating the filter as the user types, rather than for searching, in which case you would wait until the user "submits" the search request.

Our implementation takes advantage of `ArrayAdapter`'s built-in filtering capability:

```
@Override
public boolean onQueryTextChange(String newText) {
  if (TextUtils.isEmpty(newText)) {
    adapter.getFilter().filter("");
  }
  else {
    adapter.getFilter().filter(newText.toString());
  }
```

```
    return(true);
  }
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

Adapters that implement the `Filterable` interface can be filtered, automatically restricting the displayed items to ones that match the filter. Calling `getFilter()` on a `Filterable` returns a `Filter`. The default implementation of a `Filter` filters on the leading characters of `toString()` of `getItem()` from the `Adapter`. Hence, filtering an `ArrayAdapter` on our roster of 25 words, where the filter string is `'m'`, would show `morbi` and `molllis` but skip `amet`, let alone other words not beginning with `m`.

So, our `onQueryTextChange()` method simply updates the `Filter` with whatever the user has typed into the `SearchView`, setting the filter to the empty string if the `SearchView` is either empty or has `null` contents.

onQueryTextSubmit()

The `onQueryTextSubmit()` method — required by `SearchView.OnQueryTextListener` – would be called if the user tapped on the submit button within the expanded `SearchView`, to ask us to perform the search. In this sample, we have disabled that button, as we are filtering our list on the fly, rather than performing a query once the `SearchView` is filled out. Hence, `ActionBarFragment` has a do-nothing implementation of `onQueryTextSubmit()`, simply returning `false` to indicate that we have not consumed the event:

```
@Override
public boolean onQueryTextSubmit(String query) {
  return(false);
}
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

The chapter on advanced database techniques has a section on full-text indexing, and the sample app in that chapter demonstrates the use of the submit button in a `SearchView` and `onQueryTextSubmit()`.

onClose()

The `onClose()` method — required by `SearchView.OnCloseListener` — in theory will be called when the `SearchView` is collapsed. Here, we simply clear out the filter

that we are using to limit the contents of the `ListView`, plus return `true` to say that we have handled the event:

```
@Override
public boolean onClose() {
  adapter.getFilter().filter("");

  return(true);
}
```

(from ActionBar/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

According to the `SearchView` source code, it will only be called if:

- The query text is empty, and
- The `SearchView` is iconified by default (`setIconifiedByDefault(true)`)

In practice, not even that works very well.

Hence, if you really need to find out when the `SearchView` is collapsed, you will probably need to use the more generic `OnActionExpandListener` interface, attached to the `SearchView` via `setOnActionExpandListener()`. `onMenuItemActionCollapse()` should be called when the `SearchView` is collapsed. This also works for other types of collapsible action views, not just `SearchView`.

SearchView… From the User's Perspective

If the user taps on the search icon, then starts typing into the `SearchView`'s editing area, the `ListView` is filtered based upon the typed-in prefix:

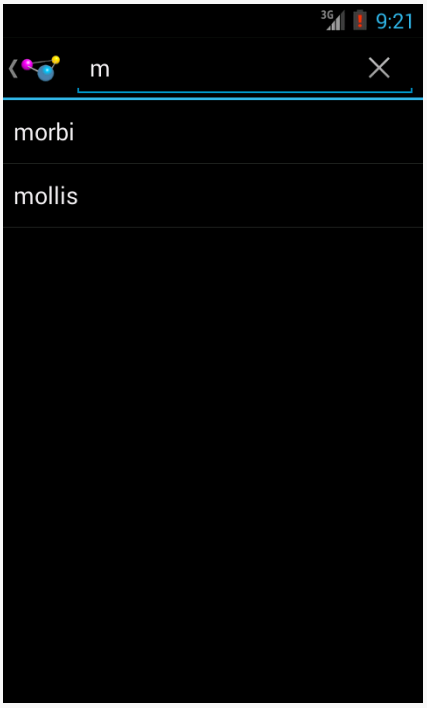


Figure 488: SearchView Demo, Showing Filtered Results

Floating Action Bars

By default, your action bar will be separate from the main content area of your activity. Normally, that is what you want.

But, sometimes, you may want to have the action bar(s) float over the top of your activity, as can be seen in Google Maps:

Figure 489: Google Maps, with Floating Action Bar (image courtesy of Google)

To accomplish this, you can use `FEATURE_ACTION_BAR_OVERLAY`, as is illustrated in the `ActionBar/OverlayNative` sample project.

This is nearly identical to the `ActionBar/ActionBarDemoNative` sample project, with just a few changes, mostly in the `onCreate()` method of our activity:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  getWindow().requestFeature(Window.FEATURE_ACTION_BAR_OVERLAY);

  initAdapter();

  Drawable d=
      getResources().getDrawable(R.drawable.action_bar_background);

  getActionBar().setBackgroundDrawable(d);
}
```

(from ActionBar/OverlayNative/app/src/main/java/com/commonsware/android/actionbaroverlay/ActionBarDemoActivity.java)

In addition to the original logic, we:

- Call `requestFeature()` on our `Window` (obtained via a call to `getWindow()`), asking for `FEATURE_ACTION_BAR_OVERLAY`
- Call `setBackgroundDrawable()` on our `ActionBar`, supplying a reference to a drawable resource to use for the background of the floating action bar

The drawable resource is a `ShapeDrawable`, defined in XML:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <solid android:color="#AAFFFFFF"/>

</shape>
```

(from ActionBar/OverlayNative/app/src/main/res/drawable/action_bar_background.xml)

We will discuss `ShapeDrawable` in much greater detail later in this book . For the moment, take it on faith that our resource is defining a rectangle, with a translucent white fill. The alpha channel (`AA`) for our translucence is important, so the user can see a bit of our activity underneath the floating action bar.

The result is that our action bars float over the top of the list:

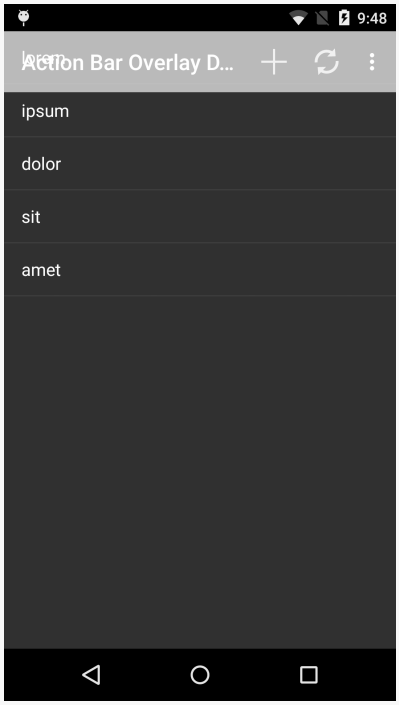


Figure 490: Floating Action Bar

In this case, the effect is not very good, as the words will blend in too strongly with the overlaid action bars. However, that is a question of organizing the screen content and using this overlay feature only in cases where you will see good results, such as in the Google Maps example shown above.

Toolbar

Android 5.0 introduced a `Toolbar` widget, offering functionality akin to the action bar, but in the form of a `ViewGroup` that can be positioned where you need it. You can even use a `Toolbar` as an outright replacement for the action bar, for cases where you need a bit more control over the action bar implementation than you get by default.

In this chapter, we will explore the use of `Toolbar`. Note that an upcoming chapter will cover the use of a backport of `Toolbar` that works back to API Level 7... albeit with some issues.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Note that the examples in this chapter are clones of a couple from the core chapters. This chapter's prose was written assuming that you were familiar with those samples, so you may need to go back and review them as needed.

One of the samples relies upon using a custom `Parcelable` class, which is covered in another chapter.

Basic Toolbar Mechanics

As noted earlier, a `Toolbar` is an ordinary `ViewGroup`. While it does not support placing arbitrary children in it the way a `LinearLayout` might, it otherwise can be used like any other `ViewGroup`. In particular, you can put it in a layout resource and

position it wherever it makes sense, such as in a lower quadrant of a tablet-sized screen, tied to some specific part of your UI.

However, the `Toolbar` is not the action bar... at least, not by default. As such, you will use somewhat different methods for interacting with it, particularly for dealing with menu items:

- You will call `inflateMenu()` when you want to pour action items into the menu, as a counterpart to the work you do in `onCreateOptionsMenu()` for the action bar
- You will call `setOnMenuItemClickListener()` to set a listener to be invoked when the user taps on a menu item in the `Toolbar`, as a counterpart to the work you do in `onOptionsItemSelected()`

A `Toolbar` does not automatically adopt much in the way of styling from your activity's theme. In particular, it does not set the background color to be the primary color of a `Theme.Material` theme, the way the action bar does. However, whether via a style resource, XML attributes in a layout file, or Java code, you can affect these same sorts of capabilities.

Use Case #1: Split Action Bar

In Android 4.x, and in the original implementation of the `appcompat-v7` action bar backport, we had the notion of the "split action bar". On phone-sized screens in portrait orientation, the action bar could easily get too crowded. We could opt into having a split action bar in these cases, where action items and the overflow would go into a bar at the bottom of the screen, leaving the top for the app's title, icon, and navigation items.

However, `Theme.Material` and modern editions of `appcompat-v7` have dropped support for the split action bar. To achieve the same basic effect, you can use a `Toolbar` that you position yourself at the bottom of the screen.

The `Toolbar/SplitActionBar` sample project demonstrates both the original Android 4.x way of getting a split action bar and using `Toolbar` to get the same basic visual effect on Android 5.0+. This is a clone of the `ActionBar/VersionedColor` sample app from a previous chapter, supporting a tinted action bar on Android 4.x (via a custom theme based off of `Theme.Holo`) and Android 5.0+ (via a custom theme based off of `Theme.Material`).

Enabling Stock Android 4.x Behavior

Getting a split action bar on Android 4.x was easy: just add `android:uiOptions="splitActionBarWhenNarrow"` to the `<activity>` or `<application>` in the manifest. Putting it on `<application>` will affect the default for all activities; putting it on a single `<activity>` affects only that activity.

The sample app's manifest uses `android:uiOptions="splitActionBarWhenNarrow"` on the one-and-only activity:

```
    <activity
      android:name="ActionBarDemoActivity"
      android:label="@string/app_name"
      android:uiOptions="splitActionBarWhenNarrow">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```

(from Toolbar/SplitActionBar/app/src/main/AndroidManifest.xml)

The result is, as the name suggests, a split action bar:

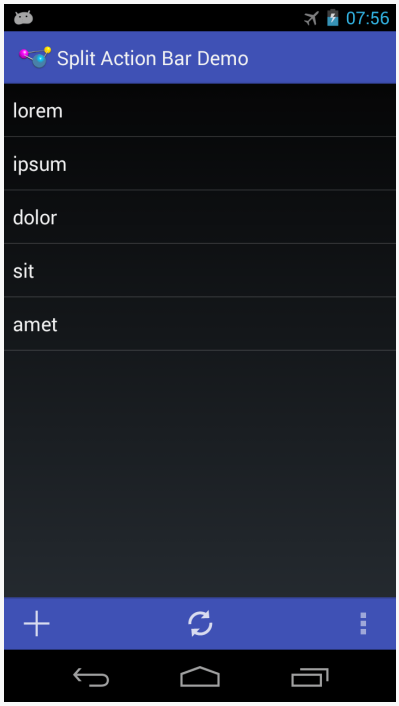


Figure 491: Split Action Bar on Android 4.3

Note that the bottom bar retains the tinting rules applied via our theme, created via the Action Bar Style Generator.

Adding the Toolbar

Since `Toolbar` is an ordinary `ViewGroup`, we can put one in a layout resource, such as `res/layout-v21/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"/>

  <Toolbar
```

```
    android:id="@+id/toolbar"
    style="@style/SplitActionBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

(from Toolbar/SplitActionBar/app/src/main/res/layout-v21/main.xml)

Here, we allocate `wrap_content` height for the `Toolbar` and give all remaining space to the `ListView` (by means of `android:layout_weight="1"` and no weight on the `Toolbar`).

The `style` attribute on the `Toolbar` points to a custom style resource, in `res/values-v21/styles.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Theme.Apptheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
  <style name="SplitActionBar">
    <item name="android:background">@color/primary</item>
  </style>
</resources>
```

(from Toolbar/SplitActionBar/app/src/main/res/values-v21/styles.xml)

This sets the background color of the `Toolbar` to be the same background color that we are using for the `colorPrimary` tint for our `Theme.Material`-based custom theme. By default, `Toolbar` has a black background, despite setting `colorPrimary` on the theme.

Using the Layout

In `onCreate()` of the activity, we load up the layout file if we are on Android 5.0 or higher:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
```

```
      setContentView(R.layout.main);
    }

    initAdapter();
  }
```

(from Toolbar/SplitActionBar/app/src/main/java/com/commonsware/android/toolbar/sab/ActionBarDemoActivity.java)

Note that we could have had a separate `res/layout/main.xml` resource, containing just the `ListView`. Then, we could call `setContentView()` regardless of API level, with the resource system pulling in the right one based on the device's API level. In this case, since we are using `ListActivity`, we do not need a layout for Android 4.x. Having two lines of Java versus a separate layout resource is a tradeoff that could be made either way.

This gives us a `Toolbar`, but by default it will be empty, making it less than useful.

Populating and Using the Toolbar

On Android 4.x, we can just implement `onCreateOptionsMenu()` and `onOptionsItemSelected()`, and the items will work, whether we chose a split action bar or not. On Android 5.0+, we need to explicitly put the action bar items into the `Toolbar` and explicitly register a listener to find out when those items are tapped.

We handle all of that in `onCreateOptionsMenu()` itself, using different behavior based on API level:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
    Toolbar tb=(Toolbar)findViewById(R.id.toolbar);

    tb.inflateMenu(R.menu.actions);
    tb.setOnMenuItemClickListener(new Toolbar.OnMenuItemClickListener() {
      @Override
      public boolean onMenuItemClick(MenuItem item) {
        return(onOptionsItemSelected(item));
      }
    });
  }
  else {
    getMenuInflater().inflate(R.menu.actions, menu);
  }
```

```
    return(super.onCreateOptionsMenu(menu));
  }
```

(from Toolbar/SplitActionBar/app/src/main/java/com/commonsware/android/toolbar/sab/ActionBarDemoActivity.java)

If we are on an Android 4.x device, we just `inflate()` a menu resource into the supplied `Menu` for the action bar. If we are on an Android 5.0+ device, we:

- Retrieve the `Toolbar` from the inflated layout
- Inflate our menu resource into the `Toolbar` via `inflateMenu()`
- Register an `OnMenuItemClickListener` with the `Toolbar`, routing the menu item click over to our `onOptionsItemSelected()` method, so we can have one common implementation of logic for handling action items that are either in the action bar or the `Toolbar`

Results and Changes

Running this sample on Android 5.0+ gives us a split "action bar" implemented as a `Toolbar`:

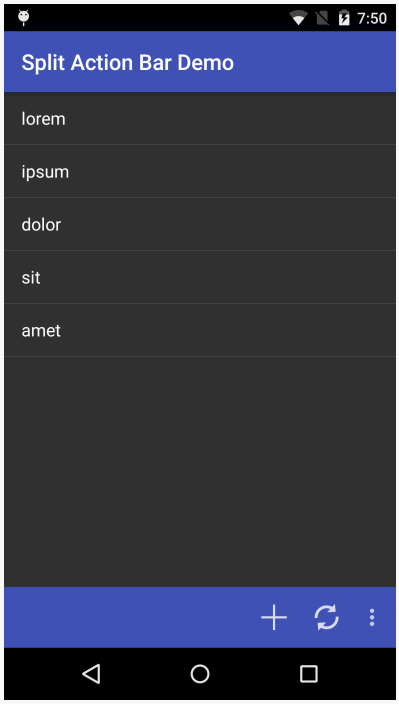


Figure 492: Split "Action Bar", Via a Toolbar, on Android 5.1

One significant visual difference is the horizontal placement of the action items. In a true split action bar, they are evenly spaced across the bar. In a `Toolbar`, they are flush right (or, more accurately, flush "end", to handle right-to-left languages). There is nothing built into `Toolbar` to spread the items out. While there are hacks to make this happen, they rely on internal implementation of `Toolbar` and may prove unreliable over time.

Use Case #2: Contextual Actions

Sometimes, the reason to consider a `Toolbar` is that you want the user to have an easier time performing actions that pertain to a part of the UI, instead of the whole UI. This is particularly the case on tablet-sized screens, where the visual gap between parts of your UI and the top action bar may be substantial.

As an example, the `Toolbar/EU4YouToolbar` sample project is based on the `EU4You` samples from the chapter on large-screen strategies. There, we had a master/detail pattern with a list of member nations of the EU as the master and the mobile Wikipedia page as the detail.

`EU4YouToolbar` makes a few changes:

- On tablets, it splits the detail area, to show a larger rendition of the country's flag, to go along with the mobile Wikipedia page.
- It adds navigational controls, so as the user browses the Web through the `WebView` in our UI, the user can go forward and backwards in their browsing history, plus reload the current page. On smaller screens, where the `WebView` fills the screen, these controls are in the action bar:

Figure 493: EU4YouToolbar Sample, on a Nexus 5

On larger screens, these controls are in a `Toolbar` placed immediately above the `WebView`:

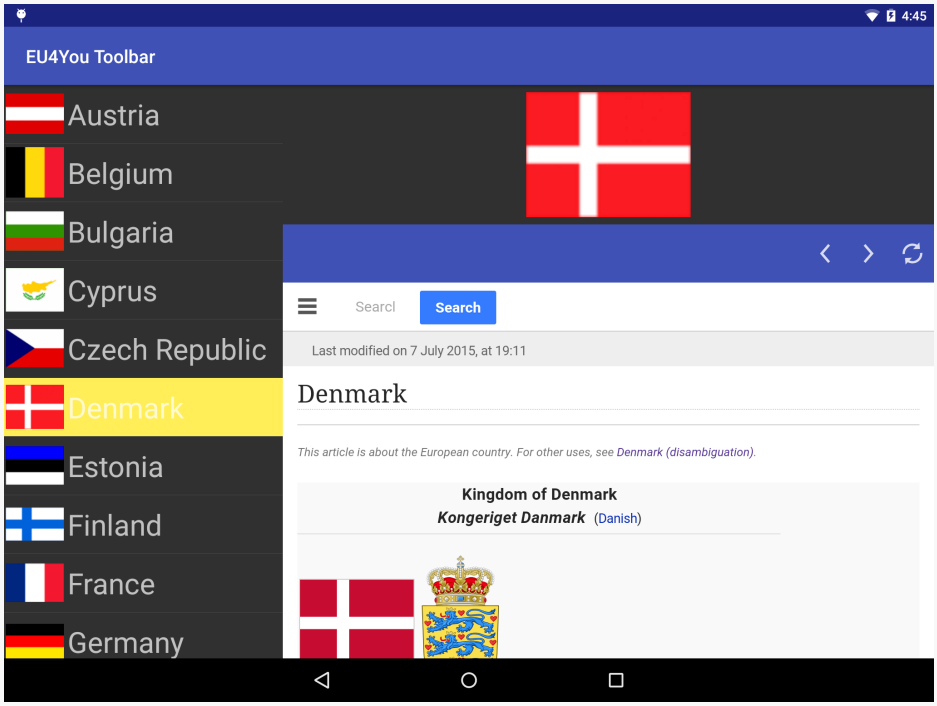


Figure 494: EU4YouToolbar Sample, on a Nexus 9

To keep things a bit simpler, this project has a `minSdkVersion` of 21, so we do not need to fuss with backwards compatibility. In truth, this would not be too difficult, requiring a different large-screen detail layout (that lacks the `Toolbar`) and falling back to having the navigational controls in the action bar if we cannot find a `Toolbar`.

The original sample used a `WebViewFragment` subclass (`DetailFragment`) to display the detail, and it supplied its own `WebView`. Now, we may want to show a flag (`ImageView`) and `Toolbar` as well, so we need our own layouts. Normally, we still only show a `WebView`:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/webview"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

</WebView>
```

(from Toolbar/EU4YouToolbar/app/src/main/res/layout/details.xml)

However, on 720dp or larger screens, we add in an `ImageView` for the flag and a `Toolbar` for the navigational controls:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ImageView
    android:id="@+id/flag"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_margin="8dp"
    android:layout_weight="1"
    android:scaleType="fitCenter"/>

  <Toolbar
    android:id="@+id/toolbar"
    style="@style/Toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <WebView
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="3"/>

</LinearLayout>
```

(from Toolbar/EU4YouToolbar/app/src/main/res/layout-w720dp/details.xml)

That layout gives the `Toolbar` a style of `@style/Toolbar`, which sets the background color of the `Toolbar` to be the primary color used by our overall theme:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Theme.Apptheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
  <style name="Toolbar">
    <item name="android:background">@color/primary</item>
```

```
  </style>
</resources>
```

(from Toolbar/EU4YouToolbar/app/src/main/res/values-v21/styles.xml)

Originally, our `DetailFragment` only needed the mobile Wikipedia URL as a data model. Now, though, we also need to know the image resource to use for the flag. While we could handle this as two separate bits of data (e.g., two extras to use with `DetailActivity`), another approach would be to pass the `Country` as the data model. However, that requires `Country` to be `Parcelable`, so we need to add some code to `Country` to fulfill the `Parcelable` contract:

```
protected Country(Parcel in) {
  name = in.readInt();
  flag = in.readInt();
  url = in.readInt();
}

@Override
public int describeContents() {
  return 0;
}

@Override
public void writeToParcel(Parcel dest, int flags) {
  dest.writeInt(name);
  dest.writeInt(flag);
  dest.writeInt(url);
}

@SuppressWarnings("unused")
public static final Parcelable.Creator<Country> CREATOR = new Parcelable.Creator<Country>() {
  @Override
  public Country createFromParcel(Parcel in) {
    return new Country(in);
  }

  @Override
  public Country[] newArray(int size) {
    return new Country[size];
  }
};
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/Country.java)

The `onCountrySelected()` method of the `EU4You` activity — which is called when the user taps on a country in the "master" list — now passes the `Country` itself over to the `DetailFragment`, whether directly or by means of starting the `DetailsActivity`:

```
@Override
public void onCountrySelected(Country c) {
```

```
    if (details != null && details.isVisible()) {
      details.showCountry(c);
    }
    else {
      Intent i=new Intent(this, DetailsActivity.class);

      i.putExtra(DetailsActivity.EXTRA_COUNTRY, c);
      startActivity(i);
    }
  }
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/EU4You.java)

`DetailsActivity` just turns around and invokes the same `showCountry()` method on `DetailsFragment` that `EU4You` uses when the `DetailsFragment` is hosted directly in `EU4You`:

```
package com.commonsware.android.eu4youtb;

import android.app.Activity;
import android.os.Bundle;

public class DetailsActivity extends Activity {
  public static final String EXTRA_COUNTRY=
      "com.commonsware.android.eu4you.EXTRA_COUNTRY";
  private Country c=null;
  private DetailsFragment details=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    details=(DetailsFragment)getFragmentManager()
                             .findFragmentById(android.R.id.content);

    if (details == null) {
      details=new DetailsFragment();

      getFragmentManager().beginTransaction()
                          .add(android.R.id.content, details)
                          .commit();
    }

    c=getIntent().getParcelableExtra(EXTRA_COUNTRY);
  }

  @Override
```

```
  public void onResume() {
    super.onResume();

    details.showCountry(c);
  }
}
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsActivity.java)

For the navigation controls, we need a menu resource. So, we define a `webview` menu resource that contains action bar items to go back in the browsing history, go forward in the browsing history, or reload the current page:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/back"
    android:title="@string/menu_back"
    android:icon="@drawable/ic_action_back"
    android:showAsAction="ifRoom"/>
  <item
    android:id="@+id/fwd"
    android:title="@string/menu_fwd"
    android:icon="@drawable/ic_action_fwd"
    android:showAsAction="ifRoom"/>
  <item
    android:id="@+id/reload"
    android:title="@string/menu_reload"
    android:icon="@drawable/ic_action_reload"
    android:showAsAction="ifRoom"/>
</menu>
```

(from Toolbar/EU4YouToolbar/app/src/main/res/menu/webview.xml)

Most of the changes, not surprisingly, reside in `DetailsFragment`, which now must manage the flag's `ImageView`, the `Toolbar` (when it exists), the action bar items (when the `Toolbar` does not exist), and the behaviors to be invoked when any of those toolbar/action bar items are invoked.

`DetailsFragment` is no longer a `WebViewFragment`, as we need our own layout. While `ListFragment` supports subclasses inflating a layout (so long as the layout has a `ListView` named `@android:id/list`), `WebViewFragment` does not. So, we inherit from the stock `Fragment` class instead and have an `onCreateView()` method that inflates our desired layout:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.details, container, false);

  webView=(WebView)result.findViewById(R.id.webview);
  flag=(ImageView)result.findViewById(R.id.flag);
  toolbar=(Toolbar)result.findViewById(R.id.toolbar);

  if (toolbar==null) {
    setHasOptionsMenu(true);
  }
  else {
    toolbar.inflateMenu(R.menu.webview);
    getNavItems(toolbar.getMenu());
    toolbar.setOnMenuItemClickListener(this);
  }

  return(result);
}
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

Here, we inflate that `details` layout resource and retrieve our three main widgets (`webView`, `flag`, and `toolbar`). However, there are two versions of that layout resource, one for larger screens and one for smaller screens. Only the larger screen has a `Toolbar`; the plan is for smaller screens to use the action bar instead. Hence, `toolbar` may be `null`.

If `toolbar` is `null`, we call `setHasOptionsMenu(true)`, to opt into this fragment participating in the action bar. If the `toolbar` is not `null`, we have it inflate a menu resource via `inflateMenu()`, and we set the fragment itself up to be the listener for click events via `setOnMenuItemClickListener()`.

In between those two steps, we call `getNavItems()`, passing the `Menu` object that the `Toolbar` is using:

```
private void getNavItems(Menu menu) {
  navBack=menu.findItem(R.id.back);
  navForward=menu.findItem(R.id.fwd);
  navReload=menu.findItem(R.id.reload);

  updateNav();
}
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

Here, we retrieve our three toolbar items, stashing them as fields in the fragment class. We also call `updateNav()`:

```
private void updateNav() {
  navBack.setEnabled(webView.canGoBack());
  navForward.setEnabled(webView.canGoForward());
  navReload.setEnabled(webView.getUrl()!=null);
}
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

`updateNav()` updates the enabled state for each of those three toolbar items, based upon the state of the `WebView`. If we can navigate back (`canGoBack()` returns `true`), we enable the back toolbar item, and so on. There is no `canReload()` method, so we substitute a check to see if the URL in the `WebView` (via `getUrl()`) is `null`.

Since we called `setOnMenuItemClickListener()` on the `Toolbar`, indicating that the fragment itself is the listener, the fragment needs to implement the `Toolbar.OnMenuItemClickListener` interface. That requires an implementation of a `onMenuItemClick()` method. In our case, as with the previous example, we delegate that to `onOptionsItemSelected()`:

```
@Override
public boolean onMenuItemClick(MenuItem item) {
  return(onOptionsItemSelected(item));
}
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

`onOptionsItemSelected()`, along with `onCreateOptionsMenu()`, will also be used if `toolbar` was `null` and we called `setHasOptionsMenu(true)` to use the action bar. So, we have a mostly-typical implementation of those methods, where `onOptionsItemSelected()` happens to be used both for the action bar and the `Toolbar` scenarios:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.webview, menu);
  getNavItems(menu);

  super.onCreateOptionsMenu(menu, inflater);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.back:
      if (webView.canGoBack()) {
```

```
        webView.goBack();
      }
      break;

    case R.id.fwd:
      if (webView.canGoForward()) {
        webView.goForward();
      }
      break;

    case R.id.reload:
      webView.reload();
      break;

    default:
      return(super.onOptionsItemSelected(item));
  }

  return(true);
}
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

Note that in `onCreateOptionsMenu()`, we call `getNavItems()`, passing in the `Menu` supplied to `onCreateOptionsMenu()`. Hence, no matter whether we are using the action bar or a `Toolbar` to host the navigation items, we have those `MenuItem` objects as fields.

The `onOptionsItemSelected()` implementation just calls appropriate methods on `WebView` tied to the particular `MenuItem`, such as `canGoBack()` and `goBack()` if the user taps the "back" `MenuItem`.

This gives us the visual result that we want. However, with the code as shown so far, the toolbar items would not change state as the user browses in the `WebView`. Their enabled states are only set when the fragment is set up. We also need to update those states as the user browses.

To handle this, we attach a `URLHandler` subclass of `WebViewClient` to the `WebView` in the `onViewCreated()` method:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);
```

```
    webView.setWebViewClient(new URLHandler());
  }
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

(note: this work could have been done in `onCreateView()`, but some of this code was ported from a sample app that used `WebViewFragment`, where we would not have an `onCreateView()` method)

Partly, `URLHandler` is responsible for ensuring that all clicks on links keep the user within the `WebView`, via a `shouldOverrideUrlLoading()` implementation. Partly, `URLHandler` is responsible for calling `updateNav()` when it appears that the navigation state of the `WebView` has changed. Unfortunately, there is no canonical place to update those navigation items, so we hook into three methods and hope for the best: `onPageStarted()`, `onPageFinished()`, and `doUpdateVisitedHistory()`:

```
private class URLHandler extends WebViewClient {
  @Override
  public void onPageStarted(WebView view, String url, Bitmap favicon) {
    super.onPageStarted(view, url, favicon);

    updateNav();
  }

  @Override
  public void onPageFinished(WebView view, String url) {
    super.onPageFinished(view, url);

    updateNav();
  }

  @Override
  public void doUpdateVisitedHistory(WebView view, String url, boolean isReload) {
    super.doUpdateVisitedHistory(view, url, isReload);

    updateNav();
  }
```

(from Toolbar/EU4YouToolbar/app/src/main/java/com/commonsware/android/eu4youtb/DetailsFragment.java)

Now, assuming that those two hooks are sufficient, our back, forward, and reload navigation items will be enabled or disabled as appropriate as the user navigates within our app and the `WebView`.

Use Case #3: Replacement Action Bar

Another thing that you can do with a `Toolbar` is make it serve as your action bar. The net effect is that you can position your activity's action bar wherever you like,

rather than have it be anchored at the top of the screen. Also, you can control the `Toolbar` more than you can the original action bar, for things like animations. For example, if you have seen apps where the action bar slides out of the way while you are scrolling down a list, only to return when you scroll back up the list, that could be accomplished via a `Toolbar` as your action bar.

The basic mechanics of making a `Toolbar` serve as the action bar are not especially difficult. Primarily, you need to inherit from `Theme.Material.NoActionBar` (to suppress the regular action bar) and call `setActionBar()` to attach your `Toolbar` to the activity to serve as the activity's action bar. As with all `Toolbar`-specific code, this will only work on API Level 21+, though the `appcompat-v7` backport offers similar capabilities.

The `Toolbar/SplitActionBar2` sample project is a clone of the `SplitActionBar` project from earlier in this chapter, except that the `Toolbar` is set up to serve as the activity's action bar.

Our activity's theme (`Theme.Apptheme`) now inherits from `Theme.Material.NoActionBar`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Theme.Apptheme" parent="android:Theme.Material.NoActionBar">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
  <style name="SplitActionBar">
    <item name="android:background">@color/primary</item>
  </style>
</resources>
```

(from Toolbar/SplitActionBar2/app/src/main/res/values-v21/styles.xml)

The `build.gradle` file sets the `minSdkVersion` to 21, so we dispense with the backwards-compatibility checks. So, in `onCreate()`, rather than conditionally using `main.xml` as our layout, we always use it, followed by a call to `setToolbar()`:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.main);
    setActionBar((Toolbar)findViewById(R.id.toolbar));
```

```
    initAdapter();
  }
```

(from Toolbar/SplitActionBar2/app/src/main/java/com/commonsware/android/toolbar/sab2/ActionBarDemoActivity.java)

Our `onCreateOptionsMenu()` can also dispense with the conditional check to see if we are on API Level 21+. However, since we are using the `Toolbar` as our action bar, we can simply populate the action bar normally, and it will affect the `Toolbar`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  return(super.onCreateOptionsMenu(menu));
}
```

(from Toolbar/SplitActionBar2/app/src/main/java/com/commonsware/android/toolbar/sab2/ActionBarDemoActivity.java)

The result is that we have a regular action bar, with its normal contents (e.g., title), but positioned where we put the `Toolbar`, at the bottom of the screen, where it used to serve as the bottom half of the split action bar:

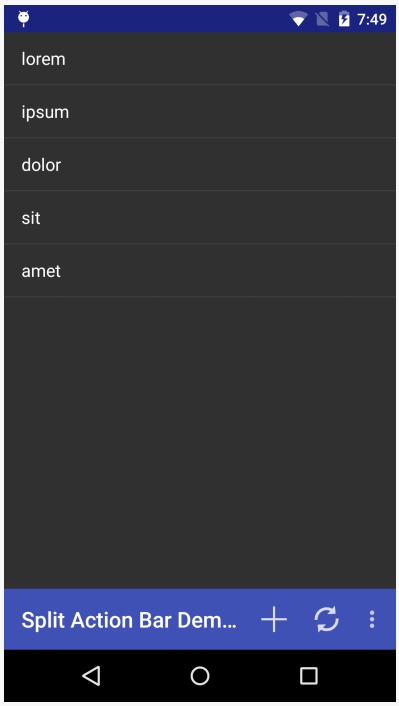


Figure 495: Toolbar as Action Bar on Android 5.1

AppCompat: The Official Action Bar Backport

Approximately 30 months after Google added the action bar to Android 3.0, Google released a backport for previous devices. Referred to here as AppCompat or, `appcompat-v7` (after its library name), this adds action bar support to Android apps, going all the way back to API Level 7.

The `appcompat-v7` Android Support Package artifact houses AppCompat. Version 21 and higher of this artifact change the way that AppCompat looks, to try to not only backport the action bar, but to backport a bit of the Material Design aesthetic.

This chapter will outline why you might want to use AppCompat and how to employ it in your Android applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Ummmm… Why?

You might wonder why we would bother with any of it. AppCompat is not required, and apps can work fine without it. And, in truth, most apps will be just fine using the native action bar implementation.

That being said, you may find some pressures nudging you towards using an action bar backport, and AppCompat specifically.

Why an Action Bar Backport?

If your `minSdkVersion` is 11 or higher, you have an action bar on all Android versions that your app supports.

If, however, your `minSdkVersion` is below 11, by default you will get the old-style options menu on Android 1.x/2.x devices. That is not a crime. However, the action bar design pattern had been used in various Android apps prior to Android 3.0's formalization of the pattern. Many apps that users will see on older devices will have an action bar, courtesy of one of the backports. By adopting a backport, you will gain a measure of consistency in your UX across Android versions that you would otherwise miss by falling back to the options menu.

You might also adopt a backport because something else is steering you to use AppCompat, and therefore you elect to use it for those reasons.

Why AppCompat?

AppCompat is a somewhat controversial library nowadays. It did not start that way when it was released in the Summer of 2013. But Google has been rather aggressive about trying to get developers to use AppCompat, and that aggressiveness has had its downsides.

Supported

The #1 reason for using AppCompat is because you decided, for other reasons, that you wanted an action bar backport, and AppCompat right now is the primary supported option.

The original backport, ActionBarSherlock, was officially deprecated by its author (Jake Wharton), who is steering you towards the native action bar or AppCompat as alternatives. While this does not prevent you from using ActionBarSherlock, it is probably not a great choice today given that Google is supporting AppCompat.

Materialistic

The current version of AppCompat does not only give you an action bar. It gives you an action bar that looks like the one that you get from `Theme.Material`. It also will attempt to apply your accent color to select widgets, the way Android 5.0 and

`Theme.Material` do, to make your app abide a bit more by the Material Design aesthetic.

Whether or not this is a good thing is up to you.

Consistent

One hidden advantage of using AppCompat, particularly in concert with the fragments backport, is consistency across Android versions. By using the native action bar and fragments, you are at some risk of inconsistent behavior based upon:

- Android OS version, due to bug fixes, deprecations, and the like
- Manufacturer or ROM modder tweaks to the native implementations, which you do not control

Having your action bar and fragments be in a library in your app isolates you from those changes. AppCompat always uses its own implementation, so any changes in the native implementation will not affect your app.

This comes at a cost of additional complexity and APK size.

Forced

Some things in the Android development ecosystem, like official support for the `MediaRouteActionProvider`, only work with the AppCompat action bar, as Google has either not shipped or has deprecated their native alternatives.

You may find some "cross-ports" of those things that work with the native action bar, but those are unlikely to be as well-supported as Google's own editions.

Also, new projects created via Android Studio basically shove `appcompat-v7` down your throat. This is why this book's tutorials have you start by importing an existing project, so you do not have to rip `appcompat-v7` and its references out by the roots to start a new project.

While it is theoretically possible that Google itself will eventually offer native action bar implementations of those things, it is unlikely. Hence, if you determine that you need one of those, you may be more inclined to use AppCompat, even if you do not need it for any other reason.

The Basics of Using AppCompat

The recipe for using the AppCompat action bar requires no new skills beyond what you have learned so far in this book. However, there are some subtle and not-so-subtle differences in the approaches AppCompat takes when compared to the native action bar.

To see the basic differences, we will take a look at the `AppCompat/ActionBar` sample project. This is a port of the fragments-and-action-bar sample from earlier in the book, where we have replaced the native action bar with AppCompat.

The Library Project

AppCompat is provided by the `appcompat-v7` Android library project, part of the Android Support Package. Just add the `implementation 'com.android.support:appcompat-v7:...'` line to your `dependencies` closure, replacing `...` with a suitable version number of the library. That will take care of downloading the library and adding it to your project.

To get the material effects described in this chapter, you will want to use version 21 or higher of `appcompat-v7` (e.g., `com.android.support:appcompat-v7:22.2.0`). And, due to a particular name change that we will examine shortly, using version 22 or higher is probably a good idea.

But, more importantly, you *really* want version 23 or higher. There are changes to ART – the Android runtime used on Android 5.0+ — that apparently will break the older versions of `appcompat-v7` when running on Android 6.0+ devices.

Your Build Settings

If you are using version 22 or higher of AppCompat, your build target must be API Level 22 or higher. Basically, for the Android Support libraries, your `compileSdkVersion` should match the major version of the library.

In Android Studio and the Android Gradle Plugin, this would be the `compileSdkVersion` found in your `build.gradle` file. In Eclipse, this would be the API level chosen in Project > Properties > Android.

Your Theme

Rather than using `Theme.Holo` or `Theme.Material`, when using AppCompat you will use `Theme.AppCompat`, whether you use that theme directly or create your own custom theme inheriting from it. There is also `Theme.AppCompat.Light` and `Theme.AppCompat.Light.DarkActionBar`, mirroring their native counterparts.

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.AppCompat">
```

(from AppCompat/ActionBar/app/src/main/AndroidManifest.xml)

Your Menu Resources

Where things start to get a bit strange with AppCompat comes with our menu resources. AppCompat forces you to use a *different namespace* for any action bar-related attributes, those added in API Level 11 or higher.

So, we started with:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/add"
    android:icon="@drawable/ic_action_new"
    android:showAsAction="always"
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@drawable/ic_action_refresh"
    android:showAsAction="always|withText"
    android:title="@string/reset"/>

</menu>
```

(from Fragments/ActionBarNative/app/src/main/res/menu/actions.xml)

and we had to change it to:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
```

```
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/add"
    android:icon="@drawable/ic_action_new"
    app:showAsAction="always"
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@drawable/ic_action_refresh"
    app:showAsAction="always|withText"
    android:title="@string/reset"/>
  <item
    android:id="@+id/about"
    android:icon="@drawable/ic_action_about"
    app:showAsAction="never"
    android:title="@string/about">
  </item>

</menu>
```

(from AppCompat/ActionBar/app/src/main/res/menu/actions.xml)

Note that we have a new `xmlns:app="http://schemas.android.com/apk/res-auto"` namespace declaration in the root `<menu>` element, and that namespace is used for the `app:showAsAction` attribute. The actual prefix name, here shown as `app`, can be whatever you want. It just has to be unique within the document and a valid XML namespace prefix (e.g., no whitespace).

Your Activity and Fragments

We have to inherit from an `AppCompatActivity` class to use AppCompat. `AppCompatActivity` itself inherits from `FragmentActivity`, and so we can use the Android Support Package's backport of fragments without issue, so you have access to backported versions of `Fragment`, `ListFragment`, etc.

NOTE: Prior to version 22 of `appcompat-v7`, you would inherit from an `ActionBarActivity` class. That class is still available for backwards compatibility, but you are recommended to inherit from `AppCompatActivity` instead.

However, note that there are no other analogues of `AppCompatActivity` for other scenarios, such as `ListActivity`. In principle, you should be able to make your own mash-ups of `AppCompatActivity` and other base activity classes, though the proof of

this is left as an exercise for the reader. The sample app just uses `AppCompatActivity` directly for showing a `ListView`:

```
package com.commonsware.android.inflation;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.Toast;
import java.util.ArrayList;

public class ActionBarDemoActivity extends AppCompatActivity {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private ArrayList<String> words=null;
  private ArrayAdapter<String> adapter=null;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.list_content_simple);
    initAdapter();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
      case R.id.add:
        addWord();

        return(true);
```

```
      case R.id.reset:
        initAdapter();

        return(true);

      case R.id.about:
        Toast.makeText(this, R.string.about_toast, Toast.LENGTH_LONG)
             .show();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }

  private void initAdapter() {
    words=new ArrayList<String>();

    for (int i=0;i<5;i++) {
      words.add(items[i]);
    }

    adapter=
        new ArrayAdapter<String>(this,
                                 android.R.layout.simple_list_item_1,
                                 words);

    setListAdapter(adapter);
  }

  private void addWord() {
    if (adapter.getCount()<items.length) {
      adapter.add(items[adapter.getCount()]);
    }
  }

  private void setListAdapter(ListAdapter la) {
    ((ListView)findViewById(android.R.id.list)).setAdapter(la);
  }
}
```

(from AppCompat/ActionBar/app/src/main/java/com/commonsware/android/inflation/ActionBarDemoActivity.java)

The layout, `res/layout/list_content_simple.xml`, is cloned from the one used by `ListActivity` itself:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
```

```
/* //device/apps/common/assets/res/layout/list_content.xml
**
** Copyright 2006, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->
<ListView xmlns:android="http://schemas.android.com/apk/res/android" android:id="@android:id/list"
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          android:drawSelectorOnTop="false"
  />
```

(from AppCompat/ActionBar/app/src/main/res/layout/list_content_simple.xml)

Your Callback Methods

In many cases, your `onCreateOptionsMenu()` and `onOptionsItemSelected()` methods will be the same for AppCompat as they would be for a regular Android app.

However, if you do need to manipulate action bar-specific attributes of your inflated menu resources, you will be unable to do so directly. The workaround is to use `MenuCompat` and `MenuItemCompat`, from the Android Support package, to give you access to newer `Menu` and `MenuItem` features in a backwards-compatible fashion.

Your Results

Visually, the results are very similar to what we get from `Theme.Material`, whether we run on Android 5.0 itself:

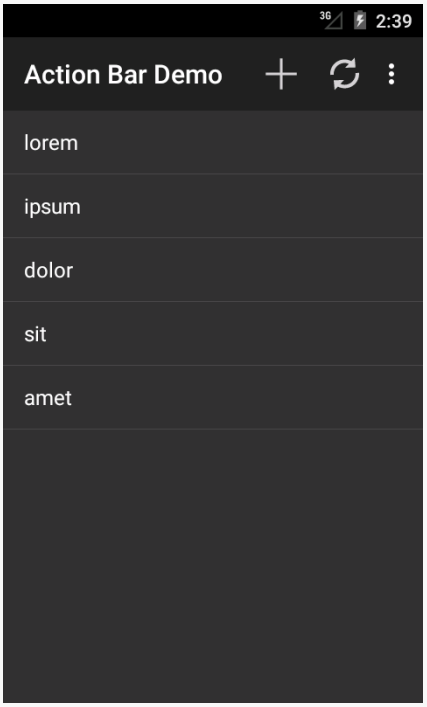


Figure 496: AppCompat, on Android 5.0 Emulator

...or on something older, like an Android 4.1 emulator:

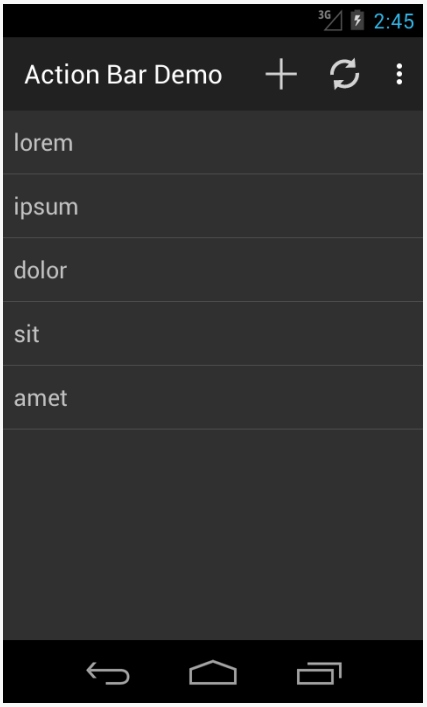


Figure 497: AppCompat, on Android 4.1 Emulator

Other AppCompat Effects

While the above recipe will give you the basics, you can go a lot further with AppCompat, just as you can with the native action bar. Generally speaking, AppCompat's backport includes all of the capabilities of the native action bar.

The biggest key is that when working with `AppCompatActivity`, if you need to access your `ActionBar` instance, call `getSupportActionBar()`, not `getActionBar()`. The latter will compile, but it will return `null` at runtime, as you are disabling the native action bar and using AppCompat's instead.

Tinting

The same basic tinting rules that apply for `Theme.Material` apply for `Theme.AppCompat` and `AppCompatActivity`, with two noteworthy differences.

First, the theme attributes do not have the `android` prefix, but instead are just the bare names (e.g., `colorPrimary`). So, in the `AppCompat/ActionBarColor` sample project, the style resource becomes:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Theme.Apptheme" parent="Theme.AppCompat">
    <item name="colorPrimary">@color/primary</item>
    <item name="colorPrimaryDark">@color/primary_dark</item>
    <item name="colorAccent">@color/accent</item>
  </style>
</resources>
```

(from AppCompat/ActionBarColor/app/src/main/res/values/styles.xml)

Second, not all widgets will have their colors affected by the theme. As of version 22 of `appcompat-v7`, the roster is limited to:

- `AutoCompleteTextView`
- `Button`
- `CheckBox`
- `CheckedTextView`
- `EditText`
- `MultiAutoCompleteTextView`
- `RadioButton`
- `RatingBar`
- `Spinner`
- `TextView`

Also, `Switch` adopts the colors, if you use the `SwitchCompat` backport discussed in the next section.

In the `AppCompat/Basic` directory you will find projects mirroring those from the `BasicMaterial` directory. In `BasicMaterial`, we saw how widgets were tinted based on a `Theme.Material`-based theme; in `AppCompat/Basic`, you will see how widgets are tinted based upon a `Theme.AppCompat`-based theme.

Switch Backport

As mentioned above, there is an official backport of the `Switch` widget, known as `SwitchCompat`, added to the `appcompat-v7` library. This works back to API Level 7, as does everything in `appcompat-v7`. And, for better or worse, it provides a backport of the Material Design implementation of a `Switch`. So, rather than the `Theme.Holo` ON/OFF toggle, we get the unlabeled "looks like a really tiny `SeekBar`" widget:

Figure 498: SwitchCompat, on an Android 4.3 Emulator

The above screenshot comes from the AppCompat/Basic/Switch sample project, which uses a layout specifying the `SwitchCompat` widget:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.SwitchCompat
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

(from AppCompat/Basic/Switch/app/src/main/res/layout/main.xml)

Also note that `SwitchCompat` only works inside an `AppCompatActivity`. It is not a general backport of `Switch` that can be used in any activity.

Overlay

AppCompat supports the same basic sort of floating action bar that is supported by the native action bar implementation. There are two slight changes in the recipe:

- Use `supportRequestWindowFeature()`, rather than `requestWindowFeature()`, to request `Window.FEATURE_ACTION_BAR_OVERLAY`
- Use `getSupportActionBar()`, rather than `getActionBar()`, to set the background

This is illustrated in `onCreate()` of the `ActionBarDemoActivity` version found in the AppCompat/Overlay sample project:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  supportRequestWindowFeature(Window.FEATURE_ACTION_BAR_OVERLAY);

  setContentView(R.layout.list_content_simple);

  initAdapter();

  Drawable d=
      getResources().getDrawable(R.drawable.action_bar_background);

  getSupportActionBar().setBackgroundDrawable(d);
  getSupportActionBar().setSplitBackgroundDrawable(d);
}
```

(from AppCompat/Overlay/app/src/main/java/com/commonsware/android/actionbaroverlay/ActionBarDemoActivity.java)

And, you get the same basic results as with the native action bar:

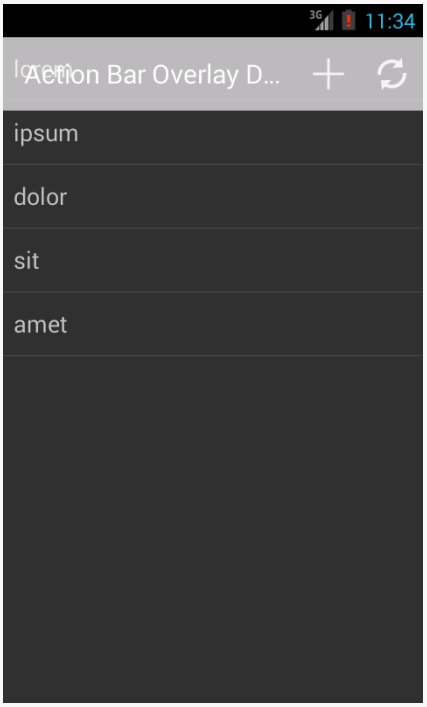


Figure 499: AppCompat with `FEATURE_ACTION_BAR_OVERLAY` on an Android 4.3 Emulator

SearchView

AppCompat has its own implementation of `SearchView`, as `android.support.v7.widget.SearchView`. To use it, switch to that class in your menu resource, then use `MenuItemCompat.getActionView()` to retrieve the instance after your menu resource has been inflated.

For example, the [AppCompat/SearchView] sample project uses the AppCompat implementation of `SearchView` in its `res/menu/actions.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/search"
    app:actionViewClass="android.support.v7.widget.SearchView"
    android:icon="@drawable/ic_action_search"
    app:showAsAction="ifRoom|collapseActionView"
```

```
    android:title="@string/filter">
  </item>

</menu>
```

(from AppCompat/SearchView/app/src/main/res/menu/actions.xml)

Then, in `onCreateOptionsMenu()`, `ActionBarFragment` calls out to a private `configureSearchView()` method that retrieves the `SearchView` and sets it up, much as you saw with the native implementation of `SearchView` from earlier in this book:

```
  private void configureSearchView(Menu menu) {
    MenuItem search=menu.findItem(R.id.search);

    sv=(SearchView)MenuItemCompat.getActionView(search);
    sv.setOnQueryTextListener(this);
    sv.setOnCloseListener(this);
    sv.setSubmitButtonEnabled(false);
    sv.setIconifiedByDefault(true);

    if (initialQuery != null) {
      sv.setIconified(false);
      search.expandActionView();
      sv.setQuery(initialQuery, true);
    }
  }
```

(from AppCompat/SearchView/app/src/main/java/com/commonsware/android/ab/search/ActionBarFragment.java)

The resulting `SearchView` is tied into AppCompat and offers a Material Design-esque look, applying your tints when opened:

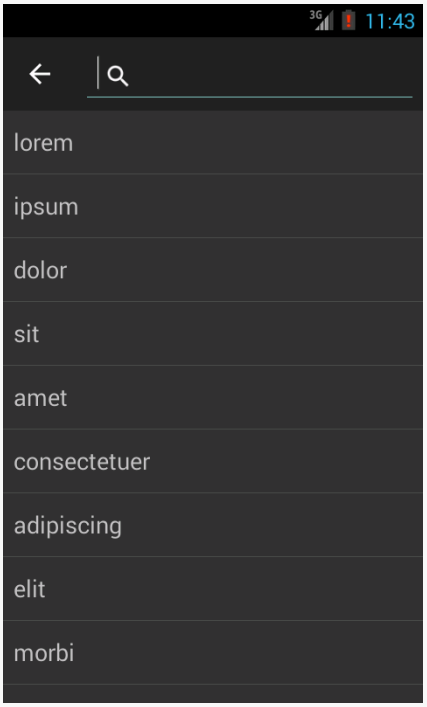


Figure 500: AppCompat with `SearchView` on an Android 4.3 Emulator

ShareActionProvider

Similarly, AppCompat has its own implementation of `ShareActionProvider`, as `android.support.v7.widget.ShareActionProvider`. The recipe for using it resembles that of using `SearchView`:

- Refer to the AppCompat edition of the class in your menu resource
- Use `MenuItemCompat.getActionProvider()` to retrieve your `ShareActionProvider` instance after inflating the menu resource, to configure and use it

The `AppCompat/Share` sample project is a clone of the `ShareActionProvider` project described elsewhere in the book, converted to use AppCompat and its edition of `ShareActionProvider`.

Toolbar and AppCompat

AppCompat has its own backport of the `Toolbar` widget. By and large, you use it in much the same way as you use the native `Toolbar`. On the plus side, the backported

`Toolbar` works back to API Level 7, allowing you to take advantage of this on much older devices. However, it requires you to be using `AppCompatActivity` — you cannot use the backported `Toolbar` with a regular activity.

The `Toolbar/SplitActionBarCompat` sample project is a clone of the `Toolbar/SplitActionBar` sample. That sample uses the native `Toolbar` to replicate the "split action bar" pattern, where there is a second "action bar" at the bottom of the screen, for actions that would not fit in the regular action bar. This clone uses the AppCompat backport of `Toolbar`, and that requires some changes.

First, we now depend upon `appcompat-v7` in the `app/` module's `build.gradle` file:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.android.support:appcompat-v7:22.2.1'
}

android {
    compileSdkVersion 22
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 14
        targetSdkVersion 22
    }
}
```

(from Toolbar/SplitActionBarCompat/app/build.gradle)

Our styles and themes change to use `Theme.AppCompat` as a base:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <style name="Theme.Apptheme" parent="Theme.AppCompat">
    <item name="colorPrimary">@color/primary</item>
    <item name="colorPrimaryDark">@color/primary_dark</item>
    <item name="colorAccent">@color/accent</item>
  </style>

  <style name="SplitActionBar">
    <item name="background">@color/primary</item>
  </style>
</resources>
```

(from Toolbar/SplitActionBarCompat/app/src/main/res/values/styles.xml)

Note that the `SplitActionBar` style, like `Theme.Apptheme`, drops the `android:` from the `name` attributes. That is because our `Toolbar` now comes from a library, and so we are no longer using system-defined attributes, but rather library-defined attributes.

The layout resource simply fully-qualifies the class name for the `Toolbar` widget to refer to the one from AppCompat:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"/>

  <android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    style="@style/SplitActionBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

(from Toolbar/SplitActionBarCompat/app/src/main/res/layout/main.xml)

`ActionBarDemoActivity` now needs to inherit from `AppCompatActivity`, rather than `ListActivity`. That means we no longer have any scenario in which we will get a `ListView` "for free" — we always have to inflate our layout resource:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.main);
    initAdapter();
  }
```

(from Toolbar/SplitActionBarCompat/app/src/main/java/com/commonsware/android/toolbar/sabc/ActionBarDemoActivity.java)

This also means we need our own `setListAdapter()` method, since we are no longer inheriting one:

```
  private void setListAdapter(ListAdapter adapter) {
    ListView lv=(ListView)findViewById(android.R.id.list);

    lv.setAdapter(adapter);
  }
```

(from Toolbar/SplitActionBarCompat/app/src/main/java/com/commonsware/android/toolbar/sabc/ActionBarDemoActivity.java)

The only other change is to the `Toolbar` `import` statement, to pull in the backport:

```
import android.support.v7.widget.Toolbar;
```

(from Toolbar/SplitActionBarCompat/app/src/main/java/com/commonsware/android/toolbar/sabc/ActionBarDemoActivity.java)

The result is visually very similar to what we would have had on Android 5.0+, but it works back to API Level 7:

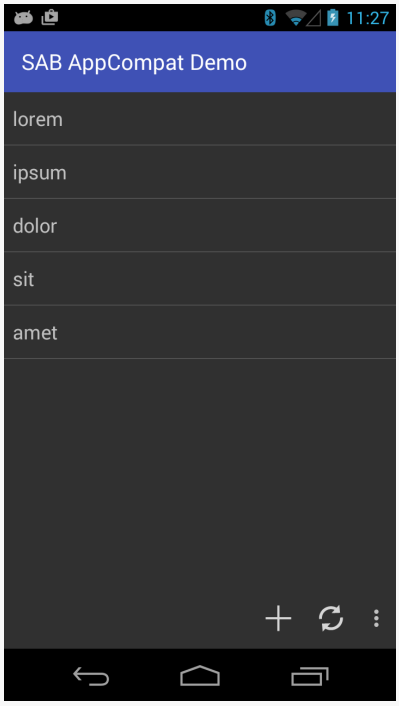


Figure 501: AppCompat Toolbar Backport, as a Split Action Bar

To Material, or Not to Material

(the following is adapted from one of the author's blog posts)

There has been a lot of discussion regarding the adoption of Material Design aesthetics in Android apps. Newcomers to Android might conclude that Material Design must be The Most Important Thing in Android Development and therefore should be pursued immediately at all costs.

Not everybody shares this opinion

With that in mind, here are the author's recommendations on what to consider with Material Design:

DO start considering the effects of Material Design upon your Android app. `Theme.Material` is the dominant theme on Android 5.0 devices, and it should remain the dominant theme on future Android versions, at least for a while. There will be hundreds of millions of these devices in use, eventually, and you will want your app to look like "it belongs" on those devices. On those devices, not only will Google's apps be employing `Theme.Material`, but manufacturer-supplied apps should do so as well. While not everything on the device may necessarily adopt this theme, more than enough will that your app may stand out somewhat if your app does not and the user thinks that it should.

DON'T blindly assume that you should be using a Material-ish look on all versions of Android. Your objective should be having an app that looks like it belongs on the device. Material-themed apps are highly unlikely to achieve majority status, let alone dominance, on pre-Android 5.0 releases. After all, device manufacturers are not going to be shipping Material-themed updates to built-in apps *en masse*, and there are plenty of apps out there that are still using pre-Holo themes. This is all on top of app that have no identifiable Android theme (e.g., most games). Certainly, there will be Material-themed apps running on Android 4.x devices, courtesy of Google and some other developers. But there will be enough Holo-themed apps that your app will not look out of place on Android 4.x any time soon, if ever. Hence, you have your choice of adopting Material Design across the board or not — user pressure is unlikely to be a major criterion any time soon.

DON'T fall victim to the "our app must look the same across all devices" mindset. Again, your app should look like it belongs on the device, which is why trying to ape an iOS look-and-feel on Android is rarely a good move. Most people do not have two

Android devices, and only a small subset of those will have an Android 5.x device and an Android 4.x (or older) device. Hence, most of the people who will care about your app appearing identical on those devices will be in your meeting room discussing the issue. Few of your users will notice — they want an app that works, does what the user wants, and looks like it belongs on their device.

DO consider whether adopting Material Design across the board may offer *engineering* benefits. For example, if you have been heavily customizing widget colors in a `Holo`-based theme, the tinting options provided by `appcompat-v7` may simplify your app a fair bit, reducing APK size, maintenance costs after the Material conversion, etc. Since your users are unlikely to be terribly concerned one way or the other, engineering considerations may help to "tip the scales" in one direction or another.

DON'T blindly assume that `appcompat-v7` will result in a simpler app, or that it should be the basis for all new apps. `appcompat-v7` has had a history of bugs. Not all of those bugs are Google's fault (e.g., the Samsung device issue was really caused by a screwy decision on Samsung's part), but that comes as cold comfort to developers trying to distribute `appcompat-v7` apps. And, if you are comfortable with using themes based on `Theme.Holo` for pre-Android 5.0 devices, `appcompat-v7` may be much more of an impediment than an advance. `appcompat-v7` is a tool, not a religion — use it where it clearly adds value *to you and your app*.

DO start planning for Google's next major theme overhaul. Google, of course, is portraying Material Design as being The One True UI Design. Google will probably get irritated when people point out that `Theme.Material` is the third major theme in the six-year production history of Android, and so assuming that `Theme.Material` is "the be-all and end-all" of themes is unrealistic. Whether the next-generation theme is a refinement on Material Design or a larger overhaul remains to be seen. But you should be taking into account that we may well wind up going through this same process in, say, early 2018. The more you can isolate the theme-related changes from the rest of your app, the more likely it is that you will be able to accommodate future theme changes with less work.

The Android Design Support Library

In 2014, to much fanfare, Google released their first edition of the Material Design guidelines.

What was missing was an actual implementation of most of these guidelines.

Beyond the obvious question of "how do you know that it will work well if you have not tried it?", it put Android developers in the unenviable position of being pressured to make their apps "look more material" without having anything really to do that.

In the months that followed Google I|O 2014, various developers took this implementation gap as a challenge and created their own implementations of many bits of Material Design. Much of this was released in the form of open source components, easily added to an app via dependencies added to a project's `build.gradle` file (at least, for Android Studio developers and other Gradle users).

In 2015, to a bit less fanfare, Google released the Android Design Support Library. The vision is that this would be the official implementation of many Material Design core components, like floating action buttons (FABs), snackbars, and the like.

This chapter explores some components from the Android Design Support Library. This chapter also explores some independent implementations of the same components, particularly ones that seem to be superior to what Google is offering at present.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar. You also should read the chapter on the `appcompat-v7` action bar backport.

Note that the examples in this chapter are clones of a couple from the core chapters. This chapter's prose was written assuming that you were familiar with those samples, so you may need to go back and review them as needed.

One of the book samples makes use of the animator framework.

GUIs and the Support Package

Many developers think that the libraries in the Android Support Package are purely backports. They then get confused when they realize that certain classes, like `ViewPager`, are not part of the core Android framework for any API level and exist only in the Android Support Package.

In truth, a lot of what is in the Android Support Package consists of backports: fragments, the action bar, `NotificationCompat`, and so on. However, the Android Support Package really consists of code that Google wants to make available to developers that can be used right away, even on older devices.

Many pieces of the Android Support Package are GUI-related, yet are not backports:

- `support-v4` and `support-v13` have the aforementioned `ViewPager`
- `cardview-v7` has `CardView`
- `recyclerview-v7` has `RecyclerView`
- `leanback-v17` has classes for "the ten-foot UI" approach used for Android apps appearing on televisions, such as via Android TV boxes

Now, we can add the Android Design Support Library to that list. Right now, this library is focused on Material Design components, and that is likely to remain its near-term focus. It remains to be seen if other GUI components, not specifically tied to Material Design, wind up in the Android Design Support Library, in `support-v4`/`support-v13`, or in other libraries.

Adding the Library… and What Comes With It

On the surface, Android Studio users can simply add `com.android.support:design:...` (for some version number for `...`, such as `25.1.1`) as a dependency:

```
implementation 'com.android.support:design:25.1.1'
```

However, this library has a transitive dependency that pulls in `appcompat-v7`. Most pieces of the Android Design Support Library do indeed seem to require that you use `appcompat-v7`, using `Theme.AppCompat`, `AppCompatActivity`, and friends. This is true even if you planned on using `Theme.Material` itself, with a `minSdkVersion` of 21 or higher.

There are two ways to work around this `appcompat-v7` requirement:

- Use a third-party implementation of the same GUI element
- Use CWAC-CrossPort, described in the next section

Introducing CWAC-CrossPort

CWAC-CrossPort is a library, published by the author of this book, that contains a subset of the GUI elements from the Design Support library. All references to `appcompat-v7` have been removed, replaced with equivalents from `Theme.Material`. As such, you can use CWAC-CrossPort on apps with a `minSdkVersion` of 21 or higher, to get the Material Design elements from the Design Support library, but without the `appcompat-v7` baggage.

CWAC-CrossPort does not include *everything* from the Design Support library, but it does include most of it.

As with the rest of the CWAC libraries, using it requires that you add the CWAC artifact repository to your Gradle configuration:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}
```

Then, add a dependency on the artifact itself, for some version:

```
dependencies {
    implementation 'com.commonsware.cwac:crossport:0.1.0'
}
```

Snackbars: Sweeter than Toasts

The `Toast` has been in Android since the beginning. It allows you to pop up a message to show the user, one that does not interfere with the rest of your activity layout. And, it is fairly easy to use.

However, some people get burned by `Toast`:

- A `Toast` is modeless, so you cannot get user input via a `Toast`
- Because a `Toast` is modeless, it is time-limited, and therefore the user might never see your message, because the user is not glancing at the screen during the short window your `Toast` is visible
- A `Toast` is a separate window from the window that is displaying your activity, so your `Toast` will remain visible even if the user navigates to some other activity, which can be annoying at times

The Material Design guidelines instead call for the use of a "snackbar", and the Design Support Library offers a `Snackbar` implementation of this UI pattern. In contrast to a `Toast`:

- A `Snackbar` is part of your activity's UI, and so it can collect input from the user, while it is around, usually in the form of some sort of "action"
- A `Snackbar` can be time-limited (for information notices) or durable (for errors or getting user input)
- While a `Snackbar` is part of your activity (and will go away when the user leaves your activity), you do not have to declare it in your layout files

With that in mind, let's take a look at some use cases for a `Snackbar` and how they can be implemented.

Alerts

The quintessential reason to use a `Toast` was to display a simple message to the user. You can use a `Snackbar` in the same role, with most of the same code.

`Snackbar` has a static `make()` method, mirroring the `makeText()` method on `Toast`. `make()` takes three parameters, only slightly different from those on `makeText()`:

1. A `View` in the activity that wishes to show the `Snackbar`
2. The message, in the form of a `CharSequence` (e.g., a `String`) or a string resource ID
3. The duration of the `Snackbar`, which is either `Snackbar.LENGTH_SHORT`, `Snackbar.LENGTH_LONG`, or possibly something else (the documentation is inconsistent on this point)

As with `makeText()` on `Toast`, simply calling `make()` on `Snackbar` creates a `Snackbar` object for you, but does not display anything. You need to call `show()` on the `Snackbar` instance to get it to appear.

The DesignSupport/Snackbar sample project is a clone of the `Threads/AsyncDemo` sample from earlier in the book. The app shows a list of 25 Latin words, progressively added to the list via an `AsyncTask`. When the list is fully populated, the original sample would display a `Toast`, from `onPostExecute()` of the `AsyncTask`.

The revised sample substitutes a `Snackbar`:

```
  @Override
  protected void onPostExecute(Void unused) {
    Snackbar.make(getListView(), R.string.done,
        Snackbar.LENGTH_LONG).show();

    task=null;
  }
```

(from DesignSupport/Snackbar/app/src/main/java/com/commonsware/android/snackbar/async/AsyncDemoFragment.java)

On phone-sized screens, the `Snackbar` will be centered along the bottom:

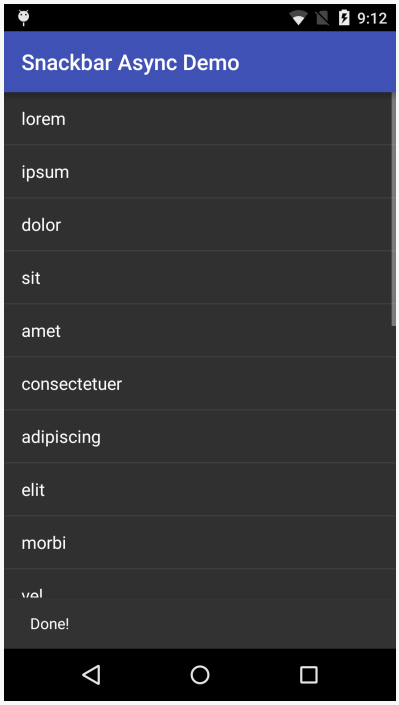


Figure 502: Official Snackbar, on a Nexus 5

On tablet-sized screens, the `Snackbar` will appear at the bottom but off to the side:

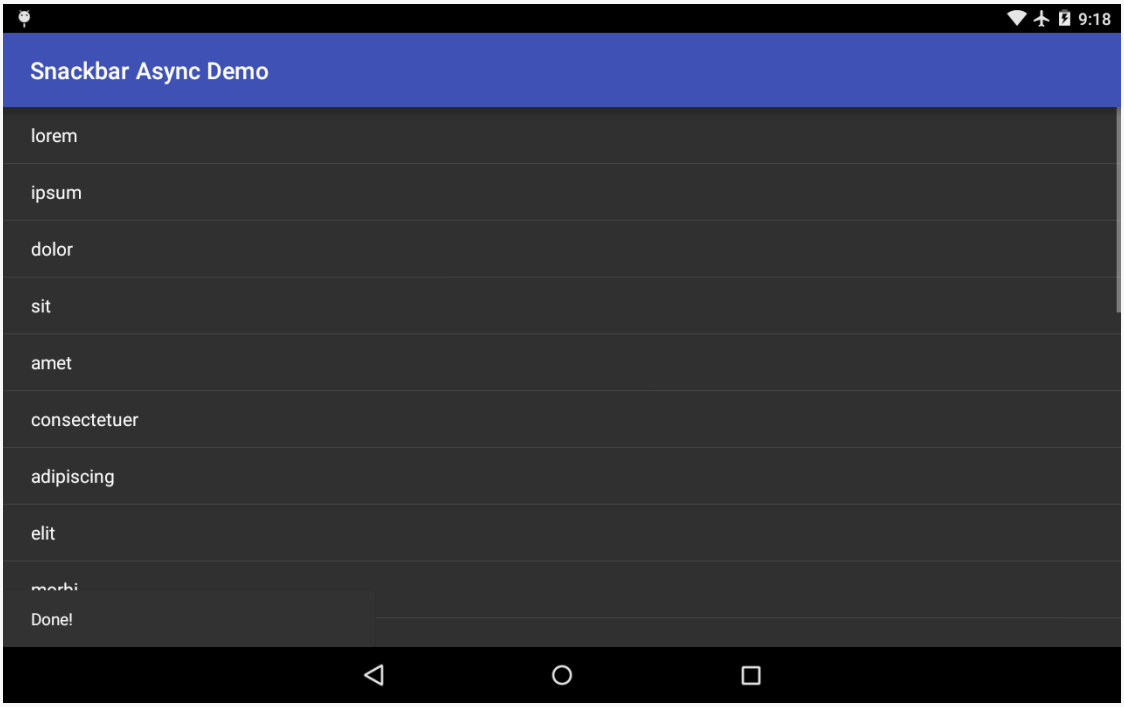


Figure 503: Official Snackbar, on a Nexus 7

Unfortunately, there does not seem to be much support for styling the look of the Snackbar. To do this manually, you can obtain the actual `View` for the `Snackbar` via `getView()`. While you should make few assumptions about what this `View` actually is, you should be able to call setters on that `View` to change things like background colors.

Also note that the user can get rid of a `Snackbar` via a swipe gesture, in addition to allowing the `Snackbar` to time out on its own. This is not possible with `Toast`, as a `Toast` is modeless.

Action Bars. No, Not Those Action Bars.

We can expand upon the user interaction with a `Snackbar` by adding an action to it. To do this, just call `setAction()` on the `Snackbar` after creating it, passing in the display string for the action (what the user will see on the `Snackbar`) and a `View.OnClickListener` that will get control when the user taps on that action. The look and feel of the action is up to the `Snackbar` implementation.

The DesignSupport/SnackbarAction sample project is a clone of the previous sample, adding one of these actions. Specifically, once the list is loaded, we want a

"Restart" action to clear the list and load it again. Perhaps the user found loading the list to be exciting and wishes to see it happen all over again.

To that end, we should pull out the work of loading our list into a `loadModel()` method that can be used from multiple places:

```
private void loadModel() {
  task=new AddStringTask();
  task.execute();
}
```

(from DesignSupport/SnackbarAction/app/src/main/java/com/commonsware/android/snackbar/action/AsyncDemoFragment.java)

The `onCreate()` method now delegates to `loadModel()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  adapter=
      new ArrayAdapter<String>(getActivity(),
          android.R.layout.simple_list_item_1,
          model);

  loadModel();
}
```

(from DesignSupport/SnackbarAction/app/src/main/java/com/commonsware/android/snackbar/action/AsyncDemoFragment.java)

And, more importantly for this section, we also call `loadModel()` from the `View.OnClickListener` of the action that we add to our `Snackbar`:

```
  @Override
  protected void onPostExecute(Void unused) {
    Snackbar munchie=Snackbar.make(getListView(), R.string.done,
        Snackbar.LENGTH_LONG);

    munchie.setAction(R.string.snackbar_action_restart,
      new View.OnClickListener() {
        @Override
        public void onClick(View view) {
          adapter.clear();
          loadModel();
        }
      });
```

```
    munchie.show();

    task=null;
  }
```

(from DesignSupport/SnackbarAction/app/src/main/java/com/commonsware/android/snackbar/action/AsyncDemoFragment.java)

The action will appear on the `Snackbar` itself, both on phones:

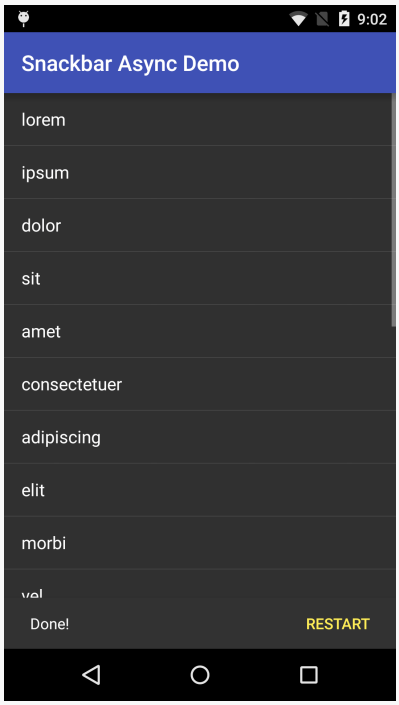


Figure 504: Official Snackbar, with an Action, on a Nexus 5

...and on tablets:

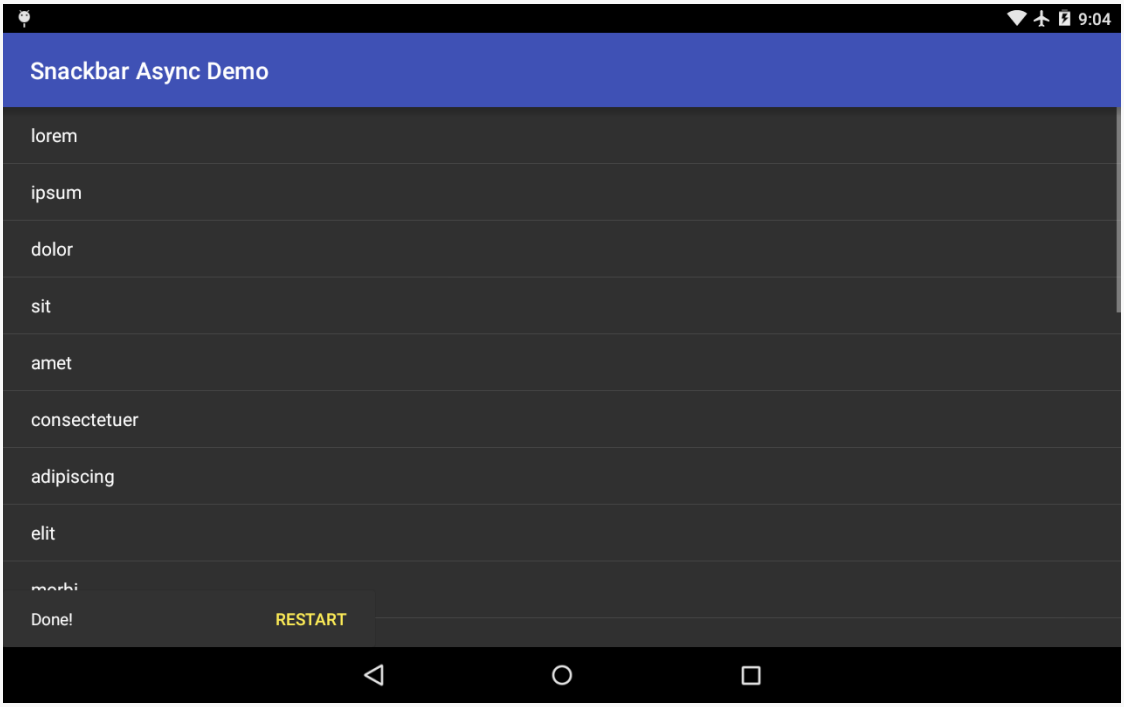


Figure 505: Official Snackbar, with an Action, on Nexus 7

Tapping the action triggers the listener, which in our case clears the list and starts the load all over again.

CWAC-CrossPort

CWAC-CrossPort supports `Snackbar`, with all the same features as are in the original Design Support implementation. The DesignSupport/SnackbarActionCP sample project is a clone of the previous sample, where we have migrated to CWAC-CrossPort.

Our `app/build.gradle` file not only adds the CWAC-CrossPort dependency, but it sets the `minSdkVersion` to 21:

```
apply plugin: 'com.android.application'

repositories {
  maven {
    url "https://s3.amazonaws.com/repo.commonsware.com"
  }
}
```

```
dependencies {
  implementation 'com.commonsware.cwac:crossport:0.2.0'
}

android {
    compileSdkVersion 26
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 26
        applicationId 'com.commonsware.android.snackbar.action.crossport'
    }
}
```

(from DesignSupport/SnackbarActionCP/app/build.gradle)

Removing `appcompat-v7` has rippling effects across the rest of the code:

- We switch to inheriting from `Theme.Material.Light.DarkActionBar` in `Theme.Apptheme`, replacing the former `Theme.AppCompat` theme
- We change the theme colors to use the `android:` prefix (e.g., `android:colorPrimary`), instead of the bare names used by `appcompat-v7`
- Our activity now just inherits from `Activity`, instead of `AppCompatActivity`
- Our fragment now just inherits from `android.app.Fragment`, instead of `android.support.v4.app.Fragment`
- Our `Snackbar` import is now for `com.commonsware.cwac.crossport.design.widget.Snackbar`

Otherwise, the code is identical, and we get identical results, though with an inverse them and adjusted title bar, to help distinguish the original from the CWAC-CrossPort edition:

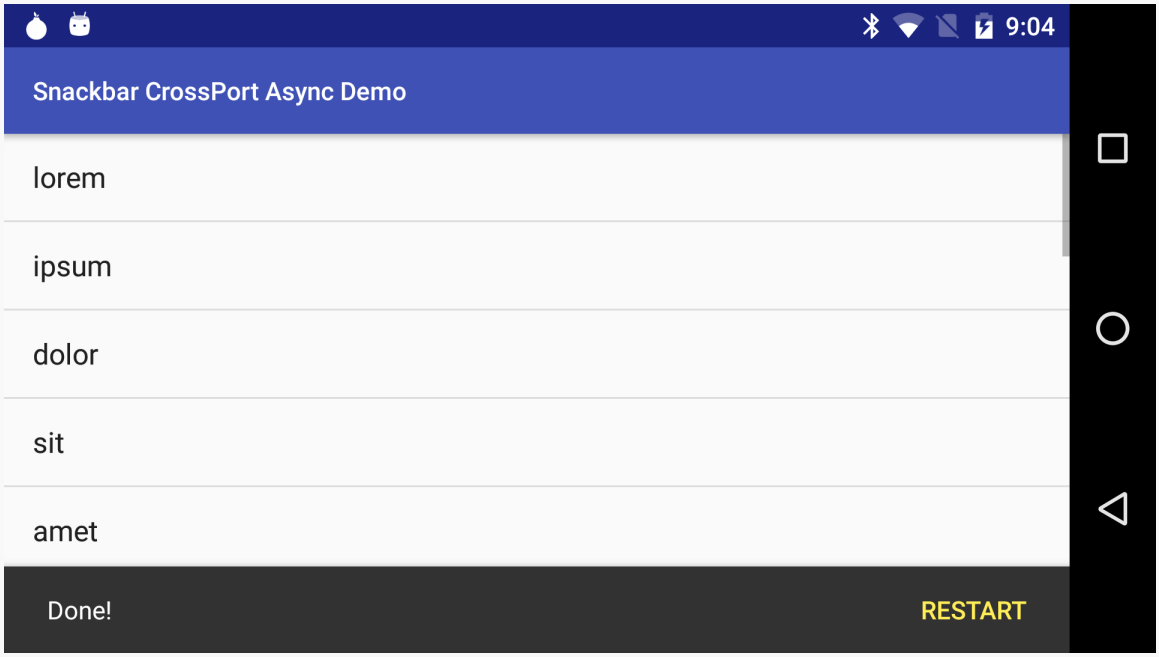


Figure 506: CWAC-CrossPort Snackbar

Absolutely FABulous

Perhaps no single element of the Material Design aesthetic has gotten more attention than has the floating action button, or FAB. These are round buttons, usually floating towards the bottom of the screen over top of the main UI:

Figure 507: Google Maps, with a Pair of FABs, on a Nexus 4

The job of the FAB is to provide rapid access to the primary action that users might take on that particular screen. Typically, in a master/detail sort of UI, the FAB will allow creating a new item for the collection:

- A voice recording app showing a list of previous recordings might have a FAB to make a new recording
- A blood pressure monitoring app showing a list of previous readings might have a FAB to take a new reading
- A to-do list app showing the current tasks might have a FAB to add a new task

However, the FAB does not have to be an "add" operation. The only real limitation is that it should be a screen-level operation, not affecting only some selected item on that screen. So, for example, in the video recording app example, you would not use a FAB to play back one of the existing videos... at least on a screen listing those videos. If tapping a video in that list brings up some sort of detail screen, *that* screen could possibly have a FAB to play back the video.

The Design Support library has a rudimentary FAB implementation, and there are third-party alternatives that either add power or solve other FAB-related problems.

FAB Mechanics

In many respects, setting up a FAB is not that different from setting up any other widget: put it in your layout, positioned where you want it, then access it in Java code to set up listeners. In particular, Google's FAB implementation supports a `View.OnClickListener` much like a regular `Button`.

The DesignSupport/FAB sample project is a clone of the first `Snackbar` sample from earlier in this chapter. The second `Snackbar` sample added a "restart" action to the `Snackbar`. In this sample, we instead have a "restart" action on a FAB.

Before, we did not need a layout resource for the `AsyncDemoFragment`, as it was an ordinary `ListFragment` and therefore would supply a `ListView` automatically. However, this time, we want to have a FAB as well, so we need our own layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
    />

  <android.support.design.widget.FloatingActionButton
    android:id="@+id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_refresh_black_24dp"
    android:layout_marginRight="@dimen/fab_margin"
    android:layout_marginBottom="@dimen/fab_margin"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"/>

</RelativeLayout>
```

(from DesignSupport/FAB/app/src/main/res/layout/main.xml)

Here, the FAB (a.k.a., `android.support.design.widget.FloatingActionButton`) is later in a `RelativeLayout` than is the `ListView`, so the FAB will have higher elevation

and will appear to float over the `ListView`. The `android:src` attribute points to a drawable resource, much like how that attribute works on an `ImageButton`.

However, the interesting bit is the pair of margin attributes (`android:layout_marginRight` and `android:layout_marginBottom`). They point to a `fab_margin` dimension resource, one with some specific values required due to bugs (or curious implementation choices) in Google's FAB implementation.

By default, the `fab_margin` in `res/values/dimens.xml` is used, which has a dimension of `0dp`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="fab_margin">0dp</dimen>
</resources>
```

(from DesignSupport/FAB/app/src/main/res/values/dimens.xml)

You might think that this would cause the FAB to be slammed up against the side and bottom of the `RelativeLayout`. However, the FAB has built-in margins... on older devices.

But, for whatever reason, on API Level 21+, that automatic margin vanishes. So, we have another definition of `fab_margin`, in `res/values-v21/dimens.xml`, setting it to `16dp`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="fab_margin">16dp</dimen>
</resources>
```

(from DesignSupport/FAB/app/src/main/res/values-v21/dimens.xml)

Furthermore, Google's Material Design docs state that there should be `24dp` margin on tablets, not `16dp`. So, we have a *third* definition of `fab_margin`, in `res/values-sw720dp-v21`, to set the margin to `24dp`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="fab_margin">24dp</dimen>
</resources>
```

(from DesignSupport/FAB/app/src/main/res/values-sw720dp-v21/dimens.xml)

It is possible that a full implementation of this would need a fourth `fab_margin` value, for Android 4.x tablets, where `fab_margin` would be set to something that gives a `24dp` margin but takes into account the automatic margin that the FAB seems to have prior to API Level 21. This sample avoids this, going with the automatic margin on all tablets, regardless of API level.

The Java code is fairly straightforward, retrieving the FAB in `onViewCreated()` and hooking up a `View.OnClickListener` to the FAB, where that listener is the `AsyncDemoFragment itself:

```
@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  getListView().setScrollbarFadingEnabled(false);
  setListAdapter(adapter);

  FloatingActionButton fab=(FloatingActionButton)v.findViewById(R.id.refresh);

  fab.setOnClickListener(this);
}
```

(from DesignSupport/FAB/app/src/main/java/com/commonsware/android/fab/async/AsyncDemoFragment.java)

In `onClick()`, if the `AsyncTask` is still running, we `cancel()` it. Then, we clear the list and kick off a fresh task via the same sort of `loadModel()` method as seen in the second `Snackbar` example:

```
@Override
public void onClick(View view) {
  if (task!=null) {
    task.cancel(false);
  }

  adapter.clear();
  loadModel();
}

void loadModel() {
  task=new AddStringTask();
  task.execute();
}
```

(from DesignSupport/FAB/app/src/main/java/com/commonsware/android/fab/async/AsyncDemoFragment.java)

This gives us our FAB on Android 4.x devices:

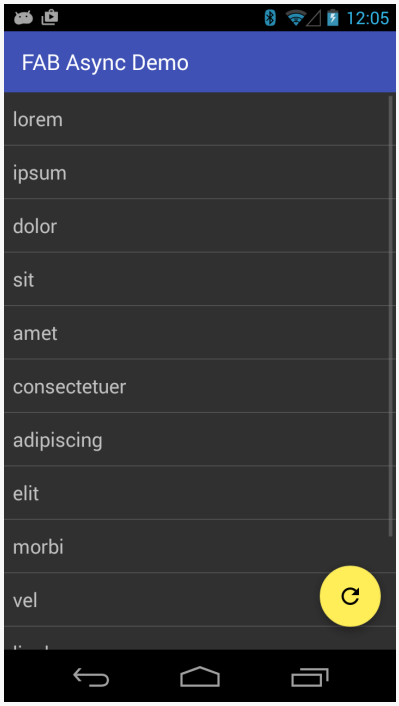


Figure 508: Official FAB, on a Galaxy Nexus

...on Android 5.x phones:

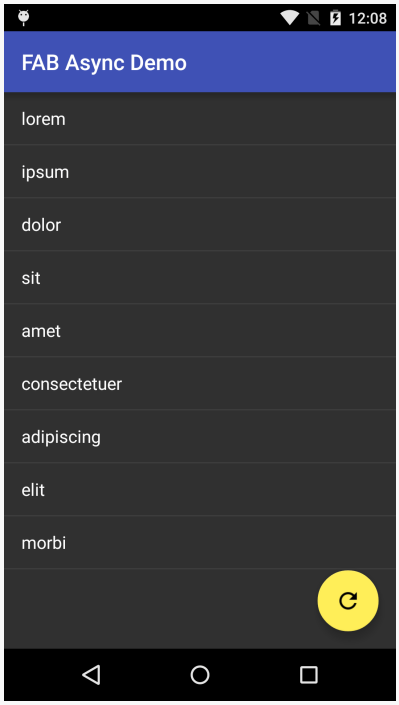


Figure 509: Official FAB, on a Nexus 5

...and on Android 5.x tablets:

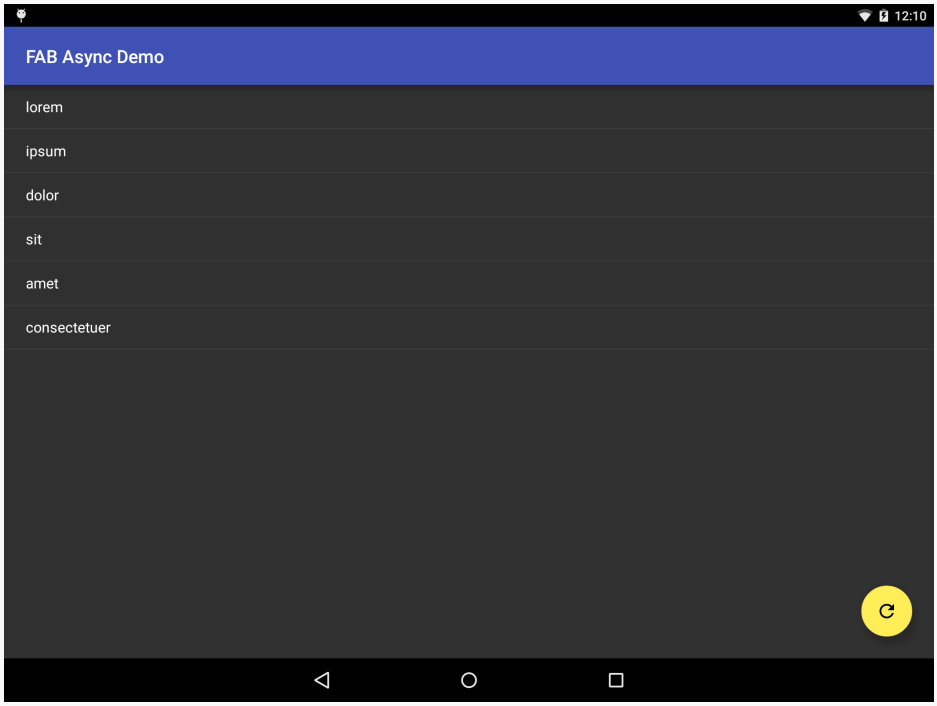


Figure 510: Official FAB, on a Nexus 9

Coordinating with Snackbars

However, what the above screenshots do *not* illustrate is what happens when our `Snackbar` appears:

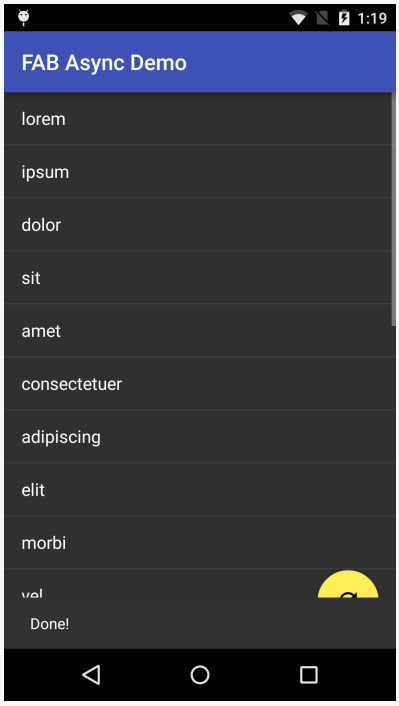


Figure 511: Official FAB, on a Nexus 5, Conflicting with the Snackbar

The fact that the `Snackbar` overlaps the FAB should not be much of a surprise. After all, the `Snackbar` overlaps the `ListView` as well. A `Toast` would also overlap the list and FAB. Hence, to some extent, the fact that there is this lack of coordination between the `Snackbar` and the FAB seems to be fairly normal.

That being said, the Design Support library has a container designed for coordinating between different children as those children animate and scroll. This container — `CoordinatorLayout` — is a subclass of `FrameLayout`, meaning other than Z-axis ordering (elevation) and gravity, it has no other notable layout rules. It merely exists to perform this sort of coordination.

As it turns out, `CoordinatorLayout` has special awareness of `Snackbar` and the FAB, so simply using `CoordinatorLayout` will cause the FAB to slide upwards to make room for the `Snackbar`.

The DesignSupport/CoordinatedFAB sample project is a clone of the previous FAB example, except that we switch from a `RelativeLayout` root container to a `CoordinatorLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
    />

  <android.support.design.widget.FloatingActionButton
    android:id="@+id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|right"
    android:layout_marginBottom="@dimen/fab_margin"
    android:layout_marginRight="@dimen/fab_margin"
    android:src="@drawable/ic_refresh_black_24dp"/>

</android.support.design.widget.CoordinatorLayout>
```

(from DesignSupport/CoordinatedFAB/app/src/main/res/layout/main.xml)

Since `CoordinatorLayout` is based on `FrameLayout`, not `RelativeLayout`, we have to adjust the layout rules on the FAB to match, using `android:layout_gravity` to position the FAB towards the bottom right corner.

With no other changes, we now get coordinated movements of the FAB and the `Snackbar` as the `Snackbar` appears and disappears:

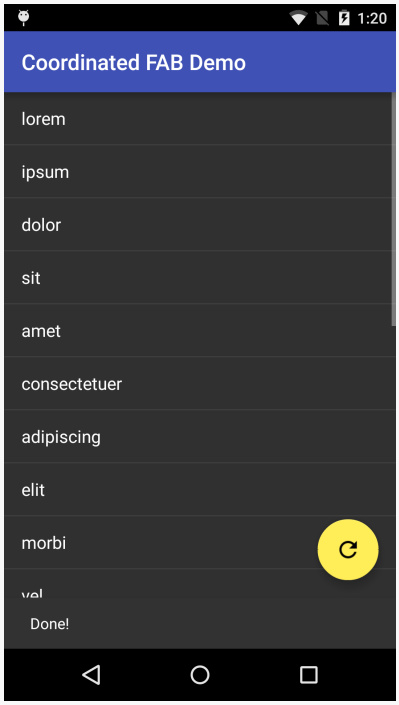


Figure 512: Official FAB, on a Nexus 5, Coordinated with the Snackbar

CWAC-CrossPort

CWAC-CrossPort also has ports of `FloatingActionButton` and `CoordinatorLayout`, as can be seen in the DesignSupport/CoordinatedFABCP sample project.

In addition to the sorts of changes needed before (e.g., replace `appcompat-v7` with `crossport`, change imports, change the theme), our layout now uses `crossport` classes:

```
<?xml version="1.0" encoding="utf-8"?>
<com.commonsware.cwac.crossport.design.widget.CoordinatorLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
    />
```

```
  <com.commonsware.cwac.crossport.design.widget.FloatingActionButton
    android:id="@+id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|right"
    android:layout_marginBottom="@dimen/fab_margin"
    android:layout_marginRight="@dimen/fab_margin"
    android:src="@drawable/ic_refresh_black_24dp"/>

</com.commonsware.cwac.crossport.design.widget.CoordinatorLayout>
```

(from DesignSupport/CoordinatedFABCP/app/src/main/res/layout/main.xml)

And, as before, we get the same results as with the official implements, just without the `appcompat-v7` overhead:

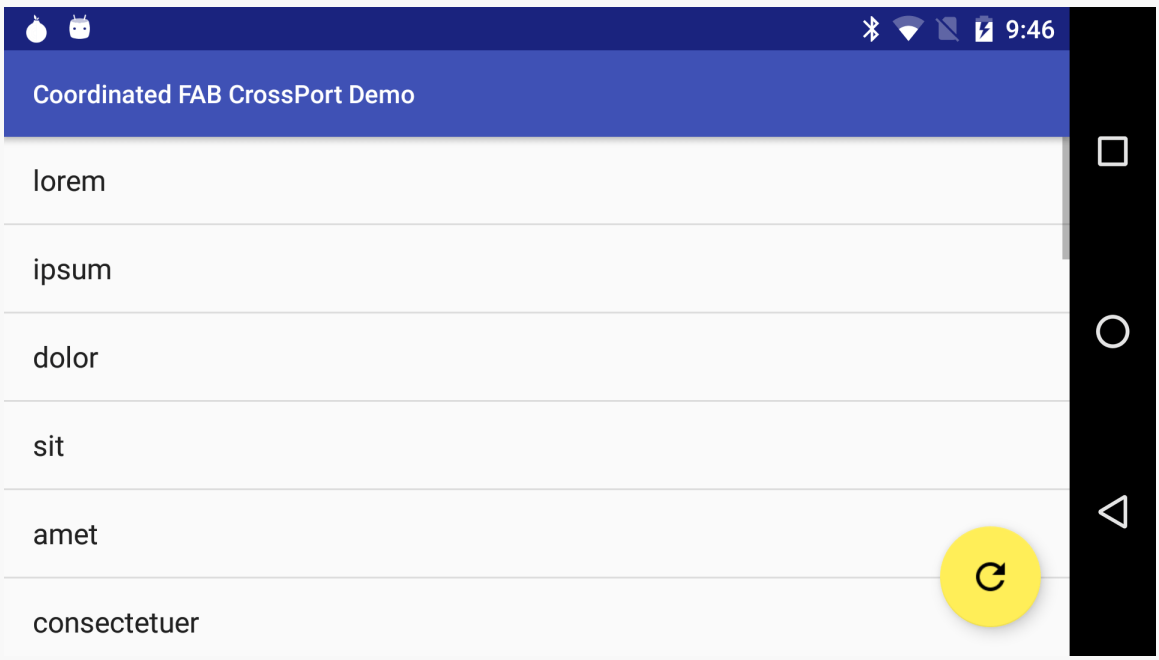


Figure 513: Coordinated FAB CrossPort Demo

Third-Party FABs… and FAMs

The Design Support implementation of a FAB works, and it works nicely "out of the box" with `CoordinatorLayout`. However, it implements only a subset of the Material Design FAB capabilities, let alone related structures like the floating action menu (FAM).

As a result, there are many FAB projects listed on the Android Arsenal, offering other implementations of a FAB. The author of this book uses the Clans FAB implementation, in part because it offers support for the FAM pattern as well:

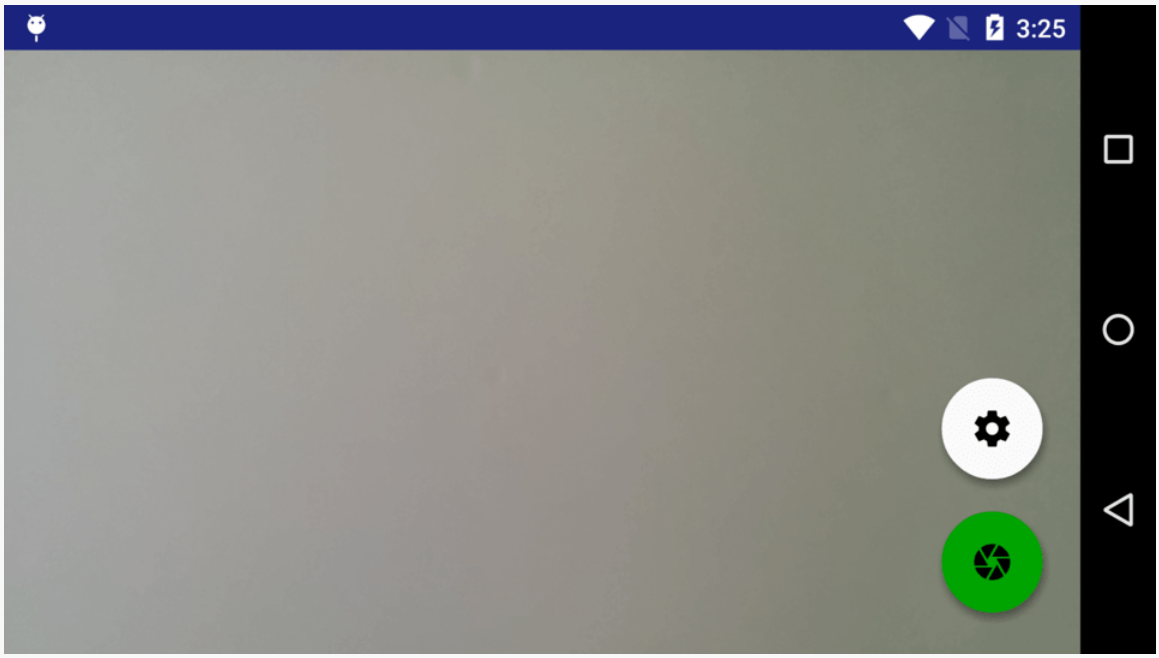


Figure 514: CWAC-Cam2 CameraActivity, Showing Clans FAB and FAM

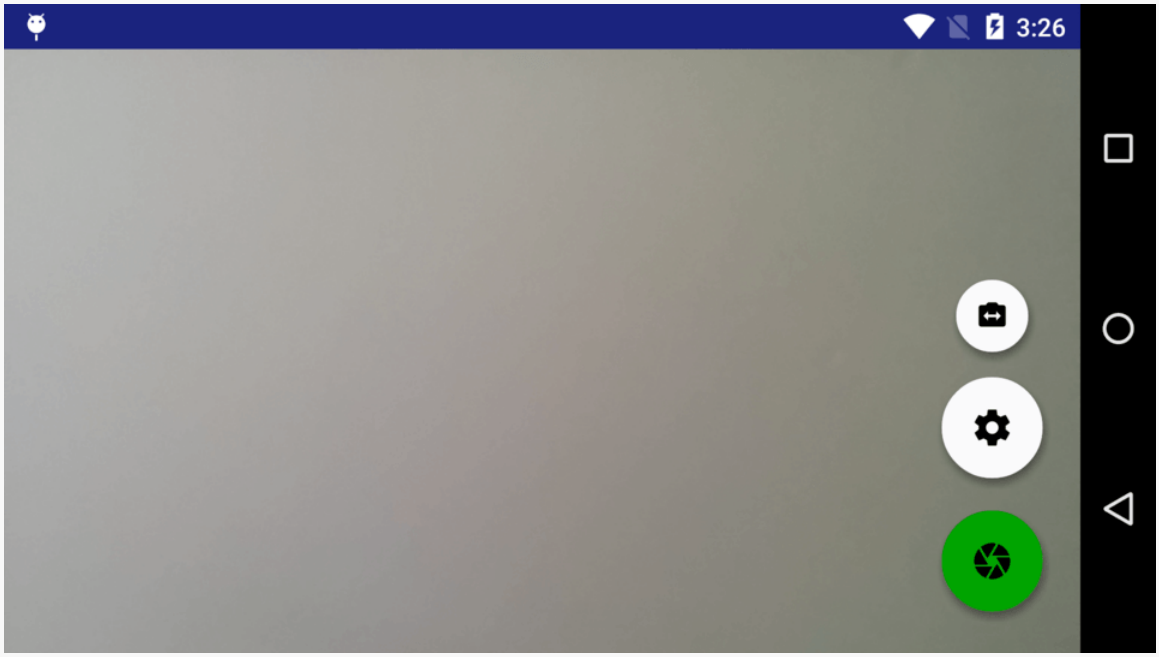


Figure 515: CWAC-Cam2 CameraActivity, Showing Clans FAB and FAM, with FAM Open

The DesignSupport/FABClans sample project is a clone of the original FAB sample shown above, where the Clans FAB implementation is used, along with a FAM.

The layout now switches to show the FAB, the FAM, and a smaller FAB that appears when the FAM is opened:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:fab="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
    />

  <com.github.clans.fab.FloatingActionButton
    android:id="@+id/refresh"
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:src="@drawable/ic_refresh_black_24dp"
    android:layout_marginBottom="@dimen/fab_margin"
    android:layout_marginRight="@dimen/fab_margin"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"/>

  <com.github.clans.fab.FloatingActionMenu
    android:id="@+id/settings"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@id/refresh"
    android:layout_alignRight="@id/refresh"
    android:layout_marginBottom="@dimen/fam_bottom_margin"
    fab:menu_colorNormal="@color/fam"
    fab:menu_colorPressed="@color/fam_pressed"
    fab:menu_icon="@drawable/ic_action_settings">

    <com.github.clans.fab.FloatingActionButton
      android:id="@+id/settings_item"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/ic_switch_camera"
      fab:fab_colorNormal="@color/fam"
      fab:fab_colorPressed="@color/fam_pressed"
      fab:fab_size="mini"/>

  </com.github.clans.fab.FloatingActionMenu>

</RelativeLayout>
```

(from DesignSupport/FABClans/app/src/main/res/layout/main.xml)

In terms of the primary FAB, the element is largely the same, other than changing the class to be `com.github.clans.fab.FloatingActionButton`.

Below that in the layout XML is a `com.github.clans.fab.FloatingActionMenu`. This is a FAB that knows how to show and hide a menu of secondary FABs when clicked. In this case, we set it to appear above the FAB with `4dp` of margin between them. In addition, we set the colors for the normal and pressed states, pointing to a pair of color resources:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#3f51b5</color>
  <color name="primary_dark">#1a237e</color>
  <color name="accent">#ffee58</color>
```

```
  <color name="fam">#fafafa</color>
  <color name="fam_pressed">#f1f1f1</color>
</resources>
```

(from DesignSupport/FABClans/app/src/main/res/values/colors.xml)

Also, for whatever reason, the icon for the FAM is not handled via the `android:src` attribute used for the FAB, but instead via a `fab:menu_icon` attribute.

The children of the FAM are supposed to be FABs that will appear and disappear as the FAM is clicked on. For this, we use `fab:fab_size="mini"` for the smaller FAB, in addition to tailoring the color.

In `onViewCreated()`, we get our hands on the main FAB and set up the `View.OnClickListener`, just as we did with the Design Support edition of the FAB. However, we also retrieve the FAM and call a `changeMenuIconAnimation()` method:

```
@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  getListView().setScrollbarFadingEnabled(false);
  setListAdapter(adapter);

  FloatingActionButton fab=(FloatingActionButton)v.findViewById(R.id.refresh);

  fab.setOnClickListener(this);

  changeMenuIconAnimation((FloatingActionMenu)v.findViewById(R.id.settings));
}
```

(from DesignSupport/FABClans/app/src/main/java/com/commonsware/android/fab/clans/AsyncDemoFragment.java)

The recommended visual effect when tapping on a FAM is not only to show the menu, but also to change the icon to a close icon, to indicate that tapping the FAM again will close the menu. Unfortunately, the Clans FAM implementation does not do this automatically. The `changeMenuIconAnimation()` method sets it up, using a combination of the Android SDK animator framework and hooks provided by the FAM for the `AnimatorSet` to be invoked when the user toggles the FAM between open and closed:

```
// based on https://goo.gl/3IUM8K

private void changeMenuIconAnimation(final FloatingActionMenu menu) {
  AnimatorSet set=new AnimatorSet();
  final ImageView v=menu.getMenuIconView();
  ObjectAnimator scaleOutX=ObjectAnimator.ofFloat(v, "scaleX", 1.0f, 0.2f);
  ObjectAnimator scaleOutY=ObjectAnimator.ofFloat(v, "scaleY", 1.0f, 0.2f);
```

```
    ObjectAnimator scaleInX=ObjectAnimator.ofFloat(v, "scaleX", 0.2f, 1.0f);
    ObjectAnimator scaleInY=ObjectAnimator.ofFloat(v, "scaleY", 0.2f, 1.0f);

    scaleOutX.setDuration(50);
    scaleOutY.setDuration(50);

    scaleInX.setDuration(150);
    scaleInY.setDuration(150);
    scaleInX.addListener(new AnimatorListenerAdapter() {
      @Override
      public void onAnimationStart(Animator animation) {
        v.setImageResource(menu.isOpened()
            ? R.drawable.ic_action_settings
            : R.drawable.ic_close);
      }
    });

    set.play(scaleOutX).with(scaleOutY);
    set.play(scaleInX).with(scaleInY).after(scaleOutX);
    set.setInterpolator(new OvershootInterpolator(2));
    menu.setIconToggleAnimatorSet(set);
  }
```

(from DesignSupport/FABClans/app/src/main/java/com/commonsware/android/fab/clans/AsyncDemoFragment.java)

Here, we:

- Create an `AnimatorSet`
- Create four `ObjectAnimator` instances, for scaling the size of something on both axes, for both scaling out (shrinking) and scaling in (growing)
- Configure the durations of each of those scale animator instances
- Add a listener to one animator to toggle the drawable used for the icon when the animation begins
- Configure the animator set to scale the old drawable out, then switch to the new icon and scale that drawable in, with an `OvershootInterpolator` for a bit of a "bounce" effect
- Tell the FAM to use that `AnimatorSet` each time the user taps the FAM

In principle, we should also be setting up a `View.OnClickListener` on the mini FAB that is inside the menu, as usually the point behind having a FAM is to respond to taps on those menu items. This is skipped in this sample in the interests of simplicity.

The resulting UI shows the FAB and the FAM, much like those from the CWAC-Cam2 library shown earlier in this section:

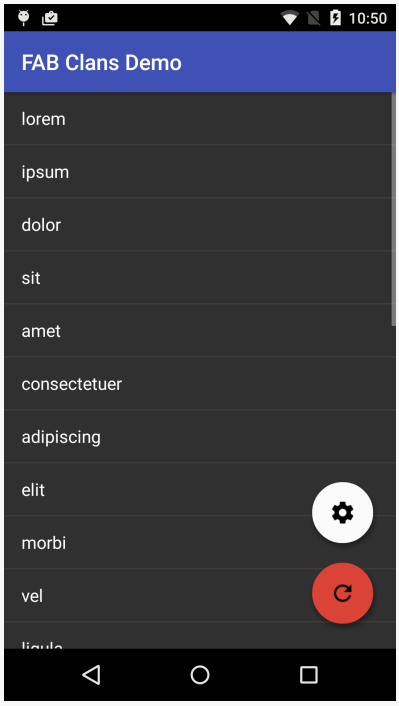


Figure 516: FABClans Sample, Showing Clans FAB and FAM

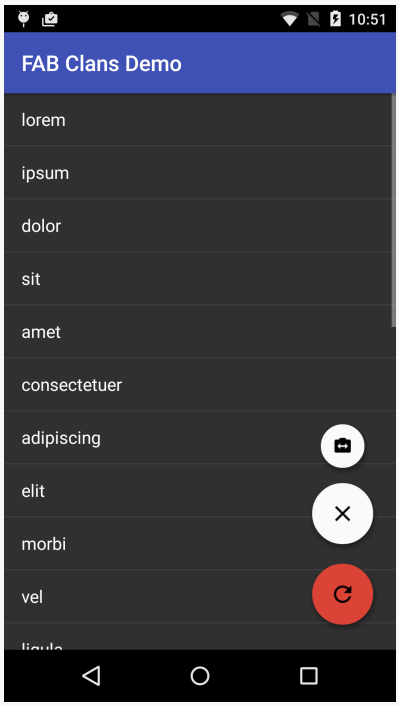


Figure 517: FABClans Sample, Showing Clans FAB and FAM, with FAM Open

Third-Party FAB Add-Ons

In addition to complete FAB and FAM implementations, developers have been publishing libraries that add on other Material Design features to existing FAB implementations, such as:

- Wrapping a progress bar around a FAB, perhaps to show the progress of some work triggered by a previous click on the FAB
- Using a "material sheet" or a "floating action toolbar" as an alternative to the FAM pattern

As usual, the Android Arsenal has a category devoted to FABs that is worth checking out.

Material Tabs with TabLayout

Android has had a myriad of tab implementations over the years:

- `TabHost` and `TabWidget`
- `FragmentTabHost` and `TabWidget`
- the now-deprecated action bar tabs
- `PagerTabStrip`, used in conjunction with a `ViewPager`

The Design Support library adds yet another tab implementation: `TabLayout`. Specifically, this implementation's claim to fame is a faithful implementation of a subset of Google's Material Design guidelines for how tabs should look and behave.

`TabLayout` can be used with or without a `ViewPager`. If you elect to skip the `ViewPager`, `TabLayout` works in a form reminiscent of action bar tabs, where it is responsible for the tab UI and you are responsible for updating the rest of your UI based upon the chosen tab (e.g., commit a `FragmentTransaction`). If you elect to use a `ViewPager`, `TabLayout` can lightly integrate with the `ViewPager`, so navigating by one means (e.g., swiping the pager) updates the other UI (e.g., changing the selected tab).

From a layout standpoint, you use `TabLayout` much like you would use `TabWidget`: put it where the tabs should go. Since Material Design wants the tabs on top, that means that typically you would put `TabLayout` inside a vertical `LinearLayout`, with the actual tabbed content beneath the `TabLayout`.

This is illustrated in the DesignSupport/TabLayout sample project. It is based on the original `ViewPager` samples, showing a set of editors in pages, this time using a `TabLayout` as the tab implementation (as opposed to `PagerTabStrip` or the `TabPagerIndicator` from the ViewPagerIndicator library).

The layout loaded by the activity has a setup much as described above: a vertical `LinearLayout` wrapped around a `TabLayout` and our `ViewPager`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">

  <android.support.design.widget.TabLayout
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>
</LinearLayout>
```

(from DesignSupport/TabLayout/app/src/main/res/layout/main.xml)

`TabLayout` can have its tabs operate in one of two modes: fixed and scrollable. With fixed tabs, all tabs will be on the screen at all times, where they divide the available horizontal space between them. This works fine for just a few tabs. But for lots of tabs, each tab becomes very small, making it unlikely that the user can read the tab caption. Scrollable tabs each take up as much room as their caption requires, and if the roster of tabs becomes too wide for the screen, the user can swipe the tabs.

The sample app demonstrates both of these approaches, using a checkable action bar item to toggle between three editors with fixed tabs or ten editors with scrollable tabs. The default state is to be in fixed-tab mode:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
  <item
    android:id="@+id/fixed"
    android:title="@string/menu_fixed"
```

```
    android:checkable="true"
    android:checked="true"
    app:showAsAction="never"/>
</menu>
```

(from DesignSupport/TabLayout/app/src/main/res/menu/actions.xml)

The entire app was switched over to use `AppCompatActivity` and the fragment backport, as that is what the Design Support library requires. Beyond that, the `EditorFragment` is pretty much unchanged from the original implementations, just showing a large `EditText` widget with a hint based on the page number.

Our `PagerAdapter` — `SamplePagerAdapter` — has one change beyond the switch to the fragment backport. To accommodate switching between fixed and scrollable tabs, rather than hard-coding the number of pages, the adapter offers a `setPageCount()` method to stipulate the number of pages. The page count defaults to 3.

```
package com.commonsware.android.tablayout;

import android.content.Context;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  private final Context ctxt;
  private int pageCount=3;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);

    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(pageCount);
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }

  @Override
```

```
  public String getPageTitle(int position) {
    return(EditorFragment.getTitle(ctxt, position));
  }

  void setPageCount(int pageCount) {
    this.pageCount=pageCount;
  }
}
```

(from DesignSupport/TabLayout/app/src/main/java/com/commonsware/android/tablayout/SampleAdapter.java)

In `MainActivity`, in `onCreate()`, we set up the `ViewPager` and the `TabLayout`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new SampleAdapter(this, getSupportFragmentManager());
    pager.setAdapter(adapter);

    tabs=(TabLayout)findViewById(R.id.tabs);
    tabs.setupWithViewPager(pager);
    tabs.setTabMode(TabLayout.MODE_FIXED);
  }
```

(from DesignSupport/TabLayout/app/src/main/java/com/commonsware/android/tablayout/MainActivity.java)

The bulk of the `TabLayout` setup work is handled with one call to `setupWithViewPager()`. This:

- Creates one tab for every page, based on whatever the `PagerAdapter` in the `ViewPager` is reporting at the time of this call
- Sets up the appropriate listeners, so that taps on a tab switches pages in the `ViewPager`, and swipes between pages update the selected tab

We also call `setTabMode(TabLayout.MODE_FIXED)`, as we are going with fixed tabs at the outset.

This gives us our three tabs:

Figure 518: TabLayout Sample, As Initially Launched, Showing Three Fixed Tabs

But we also have that menu resource, to allow the user to switch between fixed and scrollable tabs. We `inflate()` that resource in `onCreateOptionsMenu()` as usual, and we handle the checked state change in `onOptionsItemSelected()`:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.fixed) {
    item.setChecked(!item.isChecked());

    if (item.isChecked()) {
      adapter.setPageCount(3);
      tabs.setTabMode(TabLayout.MODE_FIXED);
    }
    else {
      adapter.setPageCount(10);
      tabs.setTabMode(TabLayout.MODE_SCROLLABLE);
```

```
      }

      adapter.notifyDataSetChanged();

      if (pager.getCurrentItem()>=3) {
        pager.setCurrentItem(2);
      }

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from DesignSupport/TabLayout/app/src/main/java/com/commonsware/android/tablayout/MainActivity.java)

If the user taps on our checkable overflow item, we invert the item's checked state (which, unfortunately, does not happen automatically). Then, we call `setPageCount()` on the `SampleAdapter` and `setTabMode()` on the `TabLayout` based on the now-current checked state, to either have three fixed tabs or ten scrollable tabs.

Changing the page count of the `SampleAdapter` requires calling `notifyDataSetChanged()` to alert the `ViewPager` that the data set changed and it needs to repaint. However, while the associated `TabLayout` repaints, the selected page is left alone. That is fine when going from 3 pages to 10, but it could be a problem when going from 10 pages to 3, if the selected page is after those three. So, we use `setCurrentItem()` to manually move the selection to the last valid page, if that situation occurs.

Clicking on the "Fixed" checkable overflow item, and thereby unchecking it from its initial checked state, gives us ten scrollable tabs:

Figure 519: TabLayout Sample, Showing Scrollable Tabs

Note that while this particular sample app shows `TabLayout` working with a `ViewPager`, a `ViewPager` is not required to be able to use `TabLayout`. You can simply have the `TabLayout` plus your own system for whatever the tabs switch in your UI. Then, you can use methods like `addTab()` and `setOnTabSelectedListener()` to set up tabs and find out when the user taps on them, so you can adjust your UI to match the selected tab. That being said, many users may come to expect that they can horizontally swipe to move between pages of content, and so definitely consider using a `ViewPager` if practical.

CWAC-CrossPort

CWAC-CrossPort has a port of `TabLayout`. The DesignSupport/TabLayoutCP sample project is identical to the `DesignSupport/TabLayout` sample app, except that it uses CWAC-CrossPort. The same sorts of modifications as seen earlier in this chapter are required, such as changing the layout to refer to the appropriate package for the ported `TabLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
```

```
              android:layout_height="match_parent"
              android:orientation="vertical">

  <com.commonsware.cwac.crossport.design.widget.TabLayout
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>
</LinearLayout>
```

(from DesignSupport/TabLayoutCP/app/src/main/res/layout/main.xml)

In this particular sample, since we are using menu resources, we have one additional change to make: switching from `app:showAsAction` to `android:showAsAction`, as we are using the native action bar:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
  <item
    android:id="@+id/fixed"
    android:title="@string/menu_fixed"
    android:checkable="true"
    android:checked="true"
    android:showAsAction="never"/>
</menu>
```

(from DesignSupport/TabLayoutCP/app/src/main/res/menu/actions.xml)

Third-Party Material Tabs

There are other implementations of Material Design-inspired tabs, including ones that do not have the `appcompat-v7` requirement.

One that seems to work fairly well is Karim Frenn's `MaterialTabs`. Not only can it work with native fragments and the native action bar, but it also adds features that `TabLayout` lacks, like:

- Continuous synchronization with the `ViewPager` (though there is a bug related to that, which we will examine shortly)

- More customization options, including custom fonts or even custom views for the tabs

On the other hand, `MaterialTabs` is limited to working with a `ViewPager`. It does not presently offer an API for adding tabs manually and responding to tab selection.

The DesignSupport/TabLayoutPizza sample project is so named because Mr. Frenn's GitHub account is named `pizza`. This project is a clone of the `TabLayout` sample, but it substitutes in `MaterialTabs` as the tab implementation.

In the `app/` module's `build.gradle` file, we replace the `com.android.support:design` dependency with one for `MaterialTabs` and one for `com.android.support:support-v13`. The latter is so we can use native fragments for our pages in the `ViewPager`; `MaterialTabs` itself depends upon `support-v4`.

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'io.karim:materialtabs:2.0.2'
    implementation 'com.android.support:support-v13:22.2.1'
}

android {
    compileSdkVersion 22
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 18
    }
}
```

(from DesignSupport/TabLayoutPizza/app/build.gradle)

`EditorFragment` and `SampleAdapter` are unchanged from the previous sample, except that they now use native fragments rather than the fragments backport.

The layout for `MainActivity` (`main.xml`) now has a `io.karim.MaterialTabs` widget instead of a `TabLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:app="http://schemas.android.com/apk/res-auto"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
```

```
              android:orientation="vertical">

  <io.karim.MaterialTabs
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="48dp"
    app:mtIndicatorColor="@color/accent"
    app:mtSameWeightTabs="true"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>
</LinearLayout>
```

(from DesignSupport/TabLayoutPizza/app/src/main/res/layout/main.xml)

`MaterialTabs` has a lot of custom attributes that you can set. Here, we use two:

1. `app:mtIndicatorColor` provides the color to use for the bar that indicates the selected tab.
2. `app:mtSameWeightTabs` says that if all tabs can fit on the screen, give them all the same weight, to provide the same sort of look as we get with the fixed mode with `TabLayout`. Note that `MaterialTabs` does not require manually switching modes, as it automatically switches into a scrollable mode when the number of tabs exceeds the available space.

Setting up `MaterialTabs` is then just a matter of calling `setViewPager()` on the `MaterialTabs` object to connect the tabs to the pager:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new SampleAdapter(this, getFragmentManager());
    pager.setAdapter(adapter);

    MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);
  }
```

(from DesignSupport/TabLayoutPizza/app/src/main/java/com/commonsware/android/tablayout/pizza/MainActivity.java)

Since `MaterialTabs` *does* pay attention to when the `PagerAdapter` is called with `notifyDataSetChanged()`, ideally switching between "fixed" and "scrollable" would just be a matter of setting the number of pages (via `setPageCount()`) and calling `notifyDataSetChanged()`. Unfortunately, `MaterialTabs` 2.0.2 has a bug when you reduce the number of pages, and the workaround gets a bit tricky:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.fixed) {
    item.setChecked(!item.isChecked());

    if (item.isChecked()) {
      if (pager.getCurrentItem()>2) {
        pager.setCurrentItem(2);
      }

      pager.postDelayed(new Runnable() {
        @Override
        public void run() {
          adapter.setPageCount(3);
          adapter.notifyDataSetChanged();
        }
      }, 100);
    }
    else {
      adapter.setPageCount(10);
      adapter.notifyDataSetChanged();
    }

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from DesignSupport/TabLayoutPizza/app/src/main/java/com/commonsware/android/tablayout/pizza/MainActivity.java)

`TabLayout`, when you call `setTabsFromViewPager()`, always resets your current tab to the first tab. This sucks, as the user loses her place when we add pages (or remove pages but the selected tab still exists), but at least it is reliable.

`MaterialTabs`, on the other hand, crashes if the selected tab no longer exists due to a reduction in page count. To work around this, we have to manually set the current tab to a safe tab by calling `setCurrentItem()` on the `ViewPager`, before we try changing the page count. Worse, we have to postpone the actual page count change

until after `setCurrentItem()` affects the `ViewPager` and the `MaterialTabs` objects. Hopefully, in the future, the `postDelayed()` call will no longer be needed here.

Visually, the results are pretty much the same as you get with `TabLayout`, just with the native action bar and fragments:

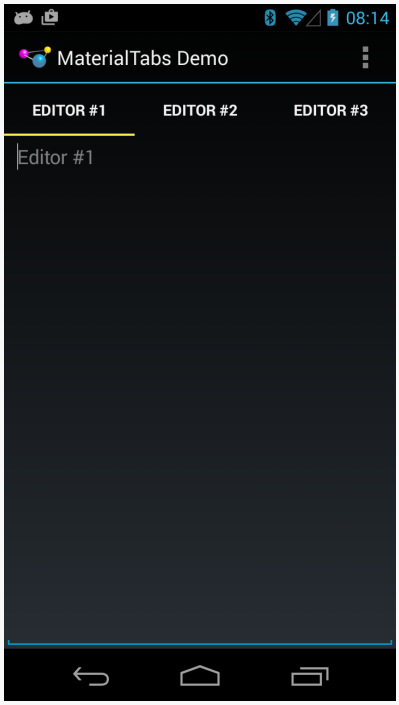


Figure 520: TabLayoutPizza Sample, As Initially Launched, Showing Three Tabs

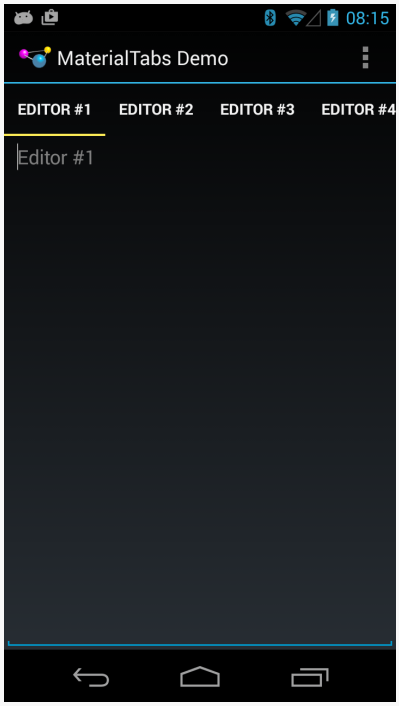


Figure 521: TabLayoutPizza Sample, As Initially Launched, Showing Ten Tabs

Floating Labels

The `EditText` widget supports the `android:hint` attribute. The hint is shown in the `EditText` when the `EditText` is otherwise empty. However, if the `EditText` has actual text in it (whether typed by the user, loaded from a database, or whatever), the hint is not shown. This saves screen space compared to having a `TextView` label always visible; the hint itself serves as the label.

However, the hint-as-label pattern has a major drawback: the hint is not visible if there is text in the `EditText`. In the long term, as the user learns your UI, this is not a big problem. However, particularly early on, the user might look at a filled-in field and wonder what that field is for. This is even more likely in cases where the user is not the one who typed the text into the field in the first place, such as editing a database entry pulled from a server, where somebody (or something) else had created the entry in the first place.

The “floating label” pattern starts with a hint in the field. However, when the field is used, the hint animates out of the field itself and “floats” above the field in a shrunken form. This way, the label is always visible. However, in its smaller floating

state, it takes up less screen space, yet while the field is otherwise empty, we can take advantage of that space to offer a "full-size" label instead.

The Design Support library offers `TextInputLayout` as a way of implementing the floating label pattern. This is not a subclass of `EditText`, but rather a `ViewGroup` that is wrapped around the `EditText`. This is convenient, insofar as it allows developers to use other `EditText` subclasses and still get the floating-label behavior.

`TextInputLayout` also supports an error state, where we can optionally show an error message below the `EditText`, such as an indication of an invalid bit of data entry.

Using TextInputLayout

The DesignSupport/FloatingLabel sample project is a clone of an earlier sample where we allowed the user to enter in a URL and then, upon a button click, would parse the URL into a `Uri`, wrap that in an `ACTION_VIEW` `Intent`, then try to start an activity for that `Intent`.

The original sample's layout looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <EditText
    android:id="@+id/url"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/url"
    android:inputType="textUri"/>

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me"/>

</LinearLayout>
```

(from Activities/LaunchWeb/app/src/main/res/layout/main.xml)

In this revised sample, the original `EditText` is augmented with a `TextInputLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me">

    <requestFocus/>
  </Button>

  <android.support.design.widget.TextInputLayout
    android:id="@+id/til"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <EditText
      android:id="@+id/url"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/url"
      android:inputType="textUri"/>
  </android.support.design.widget.TextInputLayout>

</LinearLayout>
```

(from DesignSupport/FloatingLabel/app/src/main/res/layout/main.xml)

You will notice that there are a few changes here:

- The `EditText` is wrapped by an `android.support.design.widget.TextInputLayout` container that provides the actual floating label itself
- The `Button` is moved ahead of the `EditText`, in terms of the top-down organization of our `vertical` `LinearLayout`
- The `Button` has a `<requestFocus/>` child element, indicating to Android that this widget should get the focus first

Those latter two changes are due to one major limitation with `TextInputLayout`: the hint moves out of the `EditText` into the floating position when *either* there is text in the `EditText` *or* the `EditText` gains the focus. Strangely, simply putting the `Button` before the `EditText` is insufficient, as is simply adding `<requestFocus/>` on the `Button`. Both have to be implemented to cause the `TextInputLayout` to show the hint in its default location at the outset.

The Java code is also augmented a bit from the original sample, to take advantage of the error-reporting feature of `TextInputLayout`:

```
package com.commonsware.android.design.til;

import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.support.design.widget.TextInputLayout;
import android.support.v7.app.AppCompatActivity;
import android.util.Patterns;
import android.view.View;
import android.widget.EditText;

public class LaunchDemo extends AppCompatActivity {
  private TextInputLayout til;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    til=(TextInputLayout)findViewById(R.id.til);
    til.setErrorEnabled(true);
  }

  public void showMe(View v) {
    EditText urlField=(EditText)findViewById(R.id.url);
    String url=urlField.getText().toString();

    if (Patterns.WEB_URL.matcher(url).matches()) {
      startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse(url)));
    }
    else {
      til.setError(getString(R.string.til_error));
    }
  }
}
```

(from DesignSupport/FloatingLabel/app/src/main/java/com/commonsware/android/design/til/LaunchDemo.java)

The `Patterns` class in Android contains a series of stock regular expressions. One, `WEB_URL`, is designed to see if the URL that was entered looks like a Web URL. When the user taps the button, if the pattern matches what the user entered in the field, we go ahead and try to start the activity. If not, we show an error.

To show the error, we need to do two things:

1. Up front, we call `setErrorEnabled()`, to tell `TextInputLayout` to reserve some space for an error message
2. At the point where we want to show the error, we call `setError()` on the `TextInputLayout`

When we run the app, the `TextInputLayout` leaves the hint in the `EditText` itself, as the `EditText` is empty and does not have the focus:

Figure 522: FloatingLabel Sample, As Initially Launched

Once the user taps on the field, though, the hint "floats" above the `EditText`:

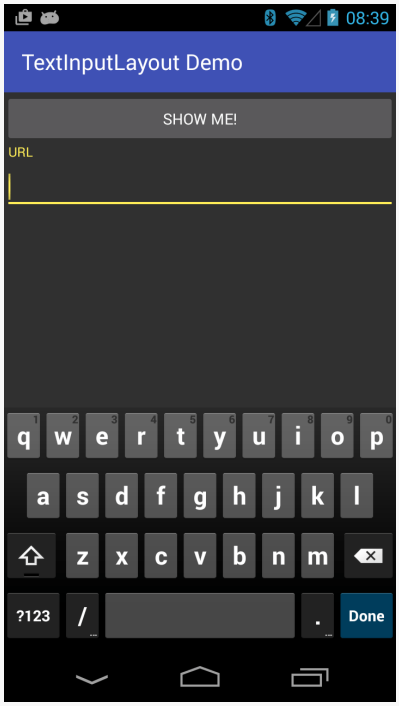


Figure 523: FloatingLabel Sample, After Focus Change

And, if the user tries entering an invalid URL, the error message appears when the user taps the button to try to visit the invalid URL:

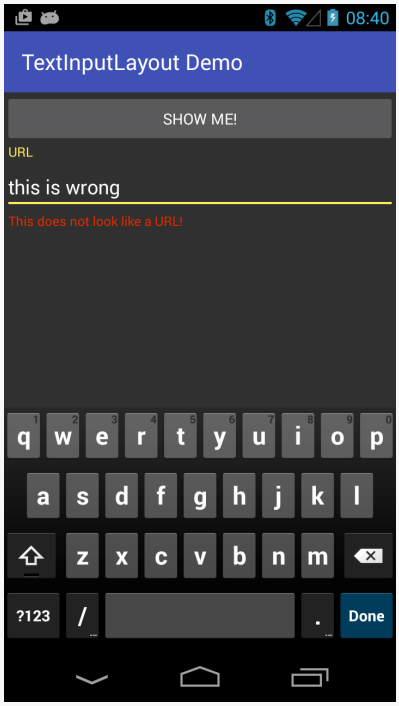


Figure 524: FloatingLabel Sample, After Erroneous Data Entry

CWAC-CrossPort

CWAC-CrossPort has a port of `TextInputLayout` and `TextInputEditText`, the latter being the `EditText` subclass that actually winds up being used with a `TextInputLayout`.

The DesignSupport/FloatingLabelCP sample project is a port of the `FloatingLabel` sample, converted to using CWAC-CrossPort. The standard sorts of changes seen earlier in this chapter, such as artifact dependencies and themes, are needed. In addition, the layout needs to change to use the CWAC-CrossPort editions of the classes:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me">

    <requestFocus />
  </Button>

  <com.commonsware.cwac.crossport.design.widget.TextInputLayout
    android:id="@+id/til"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <com.commonsware.cwac.crossport.design.widget.TextInputEditText
      android:id="@+id/url"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/url"
      android:inputType="textUri" />
  </com.commonsware.cwac.crossport.design.widget.TextInputLayout>

</LinearLayout>
```

(from DesignSupport/FloatingLabelCP/app/src/main/res/layout/main.xml)

Third-Party Floating Labels

As with the rest of the Design Support library, `TextInputLayout` requires `appcompat-v7`. There are other implementations of the floating label pattern that do not require `appcompat-v7`, or perhaps offer additional features that you may want.

`FloatLabeledEditText` is one such implementation. It lacks the error message capability of `TextInputLayout`. However:

- It only floats the hint when there is text in the `EditText` widget, not when the `EditText` gets the focus, and
- It does not require `appcompat-v7`

The DesignSupport/FloatingLabelNative sample project is a clone of the previous sample, where the `TextInputLayout` is replaced by a `FloatLabeledEditText`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">
```

```
  <com.wrapp.floatlabelededittext.FloatLabeledEditText
    android:id="@+id/til"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <EditText
      android:id="@+id/url"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/url"
      android:inputType="textUri"/>
  </com.wrapp.floatlabelededittext.FloatLabeledEditText>

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me"/>

</LinearLayout>
```

(from DesignSupport/FloatingLabelNative/app/src/main/res/layout/main.xml)

As with `TextInputLayout`, `FloatLabeledEditText` is a decorating container around a regular `EditText`. Here, since the hint is left alone when the `EditText` gets focus, we have it back in its original position at the top of the form.

Visually, it is fairly similar to `TextInputLayout`, albeit with the native action bar:

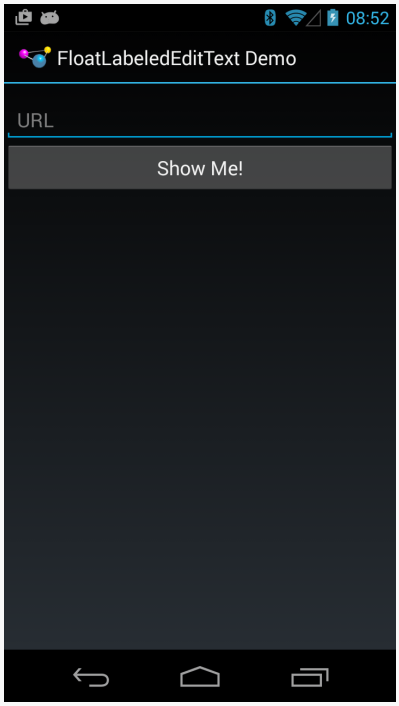
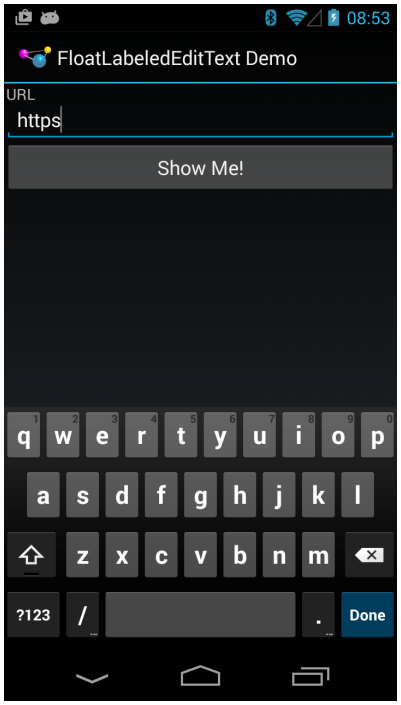


Figure 525: FloatingLabelNative Sample, As Initially Launched



Implementing a Navigation Drawer

Each year brings a new design pattern in Android that takes the development community by storm. In 2011, it was the action bar. In 2012, it was `ViewPager`. In 2013, it was the navigation drawer.

This chapter covers that navigation drawer pattern: what it is, where you use it, and how you implement it, using a `DrawerLayout` class supplied by the Android Support package.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. In addition, one section ties into the chapter on action modes.

What is a Navigation Drawer?

Complex apps often require complex navigation, to get to all of the different areas of the app. And, in many cases, that navigation is tied to nouns, reflecting different types of content, more so than verbs, reflecting operations to be performed against a particular piece of content. Verbs are actions, and can usually go in the action bar as action bar items (e.g., toolbar-style buttons). Nouns *could* be put in the action bar as well as items, though having a mixed bunch of nouns and verbs makes the action bar item roster inconsistent.

Back before the action bar, the "go-to" design pattern for navigation was the so-called "dashboard":

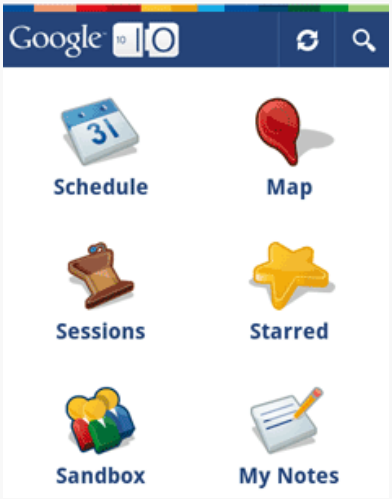


Figure 526: Google IO 2010 Conference App, with Dashboard

But this took up the whole screen and was therefore only available as the "home" activity of an app.

The navigation drawer, or "sliding menu", pattern has the same sort of content navigation options available in a drawer that slides out from the side of the screen:

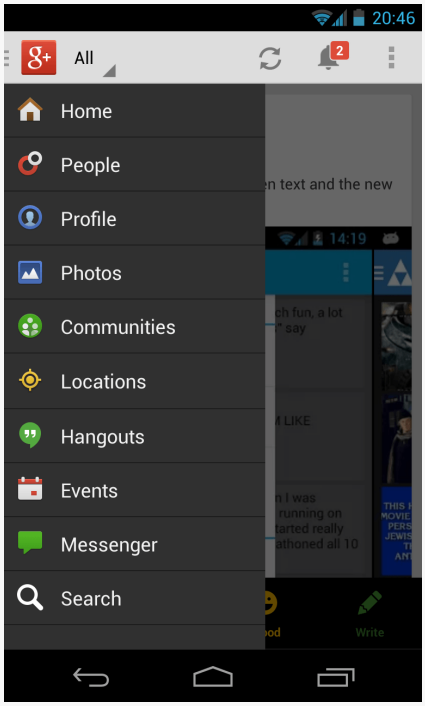


Figure 527: Google+, with Open Navigation Drawer

The drawer can be accessed from many, if not all, activities in the app, to allow the user to get wherever they need to from wherever they happen to be.

A Simple Navigation Drawer

The good news is that Google released an implementation of the navigation drawer pattern, called `DrawerLayout`, in the Android Support package (`support-v4` and `support-v13`).

The bad news is that it still takes a bit of work to get this integrated with your app.

This section will review the `NavDrawer/Simple` sample project, that shows a fairly simplistic integration of a `DrawerLayout` into an activity.

The Activity Layout

The root element of your activity layout, for an activity using `DrawerLayout`, is `DrawerLayout` itself:

```
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/drawer_layout"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <FrameLayout
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

  <ListView
    android:id="@+id/drawer"
    android:layout_width="240dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:background="#111"
    android:choiceMode="singleChoice"
    android:divider="@android:color/transparent"
    android:dividerHeight="0dp"/>

</android.support.v4.widget.DrawerLayout>
```

(from NavDrawer/Simple/app/src/main/res/layout/activity_main.xml)

`DrawerLayout` itself is rather unremarkable in the layout resource: you size and position it, usually to fill the screen. It needs to have precisely two child elements:

1. The first child represents the "real" activity content
2. The second child represents the contents of the drawer that can be opened and closed

If you are adapting an existing activity to use the `DrawerLayout`, that first child could well be an `<include>` element pointing to your existing activity layout resource, so that you can leave it undisturbed and just point your activity to start with this new `DrawerLayout` resource.

There are a couple of attributes in the children that are important for proper `DrawerLayout` operation:

- The second child — often a `ListView` — needs to have its `android:layout_gravity` set to indicate what side of the screen the drawer will slide out from. Typically this will be `left` (or `start` if you are on API Level 17+ and are taking advantage of the RTL layout support).
- The second child also specifies its `android:layout_width` to indicate the size of the drawer when opened. This should *not* be the full width of the screen, and the Android design guidelines suggest a width between 240dp and 320dp.

The ActionBarDrawerToggle

The `onCreate()` method of `MainActivity` is responsible for setting up the drawer, as well as the activity's main content:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  if (getFragmentManager().findFragmentById(R.id.content) == null) {
    showLorem();
  }

  ListView drawer=(ListView)findViewById(R.id.drawer);

  drawer.setAdapter(new ArrayAdapter<String>(
                                          this,
                                          R.layout.drawer_row,
                                          getResources().getStringArray(R.array.drawer_rows)));
  drawer.setOnItemClickListener(this);

  drawerLayout=(DrawerLayout)findViewById(R.id.drawer_layout);
  toggle=
      new ActionBarDrawerToggle(this, drawerLayout,
                                R.drawable.ic_drawer,
                                R.string.drawer_open,
                                R.string.drawer_close);
  drawerLayout.setDrawerListener(toggle);
  getActionBar().setDisplayHomeAsUpEnabled(true);
  getActionBar().setHomeButtonEnabled(true);
}
```

(from NavDrawer/Simple/app/src/main/java/com/commonsware/android/drawer/simple/MainActivity.java)

In terms of the content, if the `FrameLayout` placed in the layout resource is empty, we call `showLorem()` to lazy-create a `LoremFragment` (a `ListFragment` with 25 Latin words) and run a `FragmentTransaction` to display it:

```
private void showLorem() {
  if (lorem == null) {
    lorem=new LoremFragment();
  }

  if (!lorem.isVisible()) {
    getFragmentManager().beginTransaction()
                        .replace(R.id.content, lorem).commit();
  }
}
```

(from NavDrawer/Simple/app/src/main/java/com/commonsware/android/drawer/simple/MainActivity.java)

`onCreate()` then retrieves the `ListView`, sets the contents of the list to be a `<string-array>` resource named `drawer_rows`, and sets up the activity itself to respond to clicks on the list.

`onCreate()` then sets up an `ActionBarDrawerToggle`. This allows the app icon on the left of the action bar to open and close the navigation drawer. The `ActionBarDrawerToggle` constructor takes five parameters:

- The `Activity` as a `Context`
- The `DrawerLayout` widget to be managed by this toggle
- The icon to superimpose on the app icon to indicate that there is a toggle
- String resources for the open and close operations, for accessibility

The sample app uses icons from Google's discontinued "Action Bar Icon Pack" for the third parameter. In addition, the Android Asset Studio offers a way for you to customize the navigation drawer indicator artwork for other themes.

In addition to creating the toggle instance, we need to:

- Associate it with the `DrawerLayout`, by calling `setDrawerListener()` on it
- Enable the app icon via `setHomeButtonEnabled()` and enable it for "up" navigation via `setDisplayHomeAsUpEnabled()`
- Forward the `onPostCreate()`, `onConfigurationChanged()`, and `onOptionsItemSelected()` activity callback methods on to the toggle:

```
@Override
protected void onPostCreate(Bundle savedInstanceState) {
  super.onPostCreate(savedInstanceState);

  toggle.syncState();
}

@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);

  toggle.onConfigurationChanged(newConfig);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (toggle.onOptionsItemSelected(item)) {
    return(true);
  }
```

```
    return(super.onOptionsItemSelected(item));
  }
```

(from NavDrawer/Simple/app/src/main/java/com/commonsware/android/drawer/simple/MainActivity.java)

Without any additional work, launching the app will show the "mini hamburger" navigation drawer icon in the action bar:

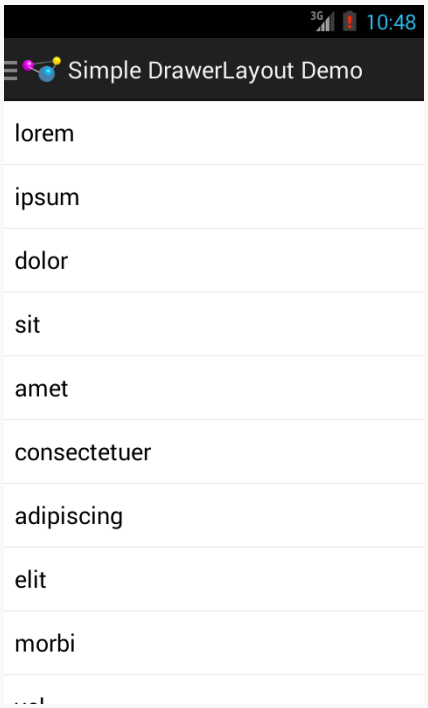


Figure 528: Nav Drawer Sample App, Showing the Mini Hamburger

Tapping the app icon will open the drawer:

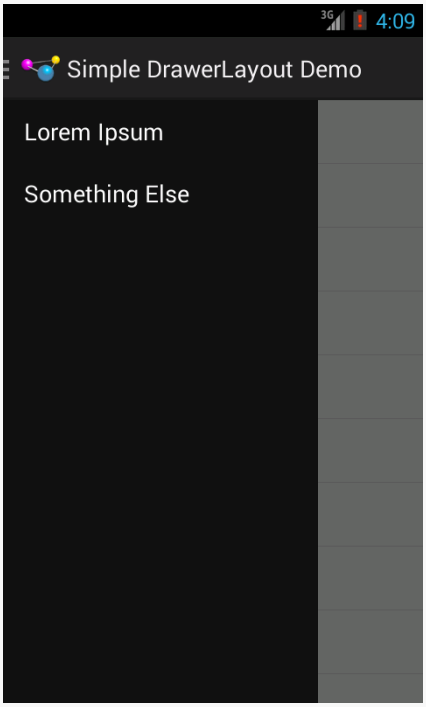


Figure 529: Nav Drawer Sample App, with Drawer Open

The drawer slides over the content, rather than pushing the content away, which is why we do not see the words peeking out on the right side of the screen.

Tapping the app icon again will close the drawer.

The user can also open the drawer via gestures. A bezel swipe from the left side will open the drawer, and swiping the open drawer right to left will close it. In addition, tapping and holding on the left edge of the content will cause the drawer to "peek" open by a handful of pixels, to hint to the user that there may be something that they can access by swiping from the left.

NOTE: This implementation of `ActionBarDrawerToggle` is officially deprecated as of the `21.0.0` version of the Android Support library. However, it is still necessary, at this time, for using `DrawerLayout` with the native action bar.

The Actions on Navigation Clicks

Of course, a navigation drawer is useless unless we do something when the user interacts with it, clicking on list rows in this case.

Our list is very simple, with just two elements. The app simply toggles between two fragments based upon the list item click:

```
@Override
public void onItemClick(AdapterView<?> listView, View row,
                        int position, long id) {
  if (position == 0) {
    showLorem();
  }
  else {
    showContent();
  }

  drawerLayout.closeDrawers();
}
```

(from NavDrawer/Simple/app/src/main/java/com/commonsware/android/drawer/simple/MainActivity.java)

We also close the drawer, using `closeDrawers()`, as otherwise the `DrawerLayout` is unaware that the user chose something and that we should return to the content.

Alternative Row Layouts

The rows in the sample app's `ListView` were fairly conventional. Rows in a nav drawer should be fairly simple, as you are merely trying to lead the users to pieces of content, not present content itself.

That being said, the navigation options can have a bit more to them than what the sample app showed. The Android design guidelines will steer you in the direction of how best to style:

- Rows with leading icons
- Rows with trailing badges (e.g., unread message counts)
- Expandable sections (e.g., less-important items but still worth having in the drawer)
- Dividers, to help organize groups of related rows

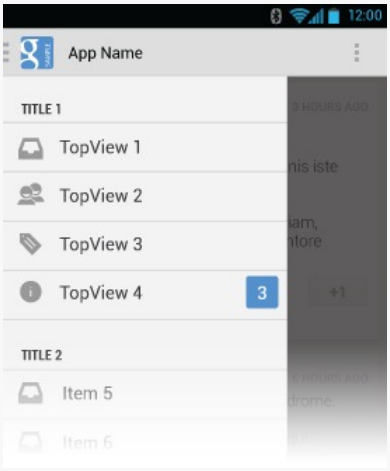


Figure 530: Sample Navigation Drawer Rows with Icons and Badges

You will see other apps experiment with other capabilities. For example, Gmail used to use `RadioButton` widgets for the accounts:

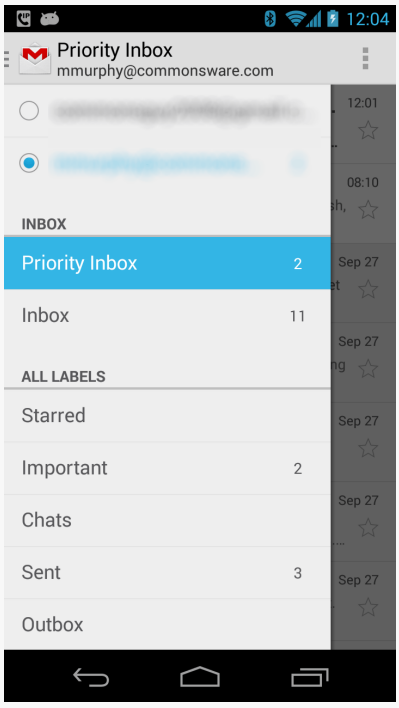


Figure 531: Gmail Navigation Drawer, with RadioButtons, Dividers, and Badges (Sans Backgrounds)

That being said, the closer you can stick with the official design guidelines, the better off you will tend to be, in terms of meeting user expectations and not encountering rendering or click event oddities with `DrawerLayout`.

Additional Considerations

Beyond putting the right things into the navigation drawer, there are some other things that you will need to take into account, particularly as your navigation drawer interacts with the rest of the activity and overall application.

Highlighting the Current Location

If the user opens the navigation drawer, and they are already at one of the navigation destinations shown in the drawer, that destination should show up with the `activated` state, to indicate to the user that she is already there. Conversely, if the user has drilled down into some part of your application that does *not* have a corresponding entry in the navigation drawer, the navigation drawer should show no activated entry.

One way to handle this is to keep the `ListView` updated as the user navigates (whether through the navigation drawer or by other means), selecting and de-selecting items as needed.

The row layout used in the original sample, culled from Google's `DrawerLayout` sample code, already has the `activated` background:

```
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/drawer_layout"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <FrameLayout
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

  <ListView
    android:id="@+id/drawer"
    android:layout_width="240dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:background="#111"
    android:choiceMode="singleChoice"
    android:divider="@android:color/transparent"
    android:dividerHeight="0dp"/>

</android.support.v4.widget.DrawerLayout>
```

(from NavDrawer/Simple/app/src/main/res/layout/activity_main.xml)

However, we were not using it in that sample, so the drawer `ListView` did not give the user any indication that they had already navigated to a navigable destination. But, we have the `NavDrawer/Activated` sample project, a clone of the original sample, that adds this activation capability, and demonstrates the headaches it causes.

First, as part of our `onCreate()` work in `MainActivity`, we need to configure the `ListView` to work in single-choice mode (the `ListView` engine behind the `activated` state), along with the rest of the `ListView` setup:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  drawer=(ListView)findViewById(R.id.drawer);
  drawer.setChoiceMode(ListView.CHOICE_MODE_SINGLE);

  String[] rows=getResources().getStringArray(R.array.drawer_rows);

  drawer.setAdapter(new ArrayAdapter<String>(this,
                                             R.layout.drawer_row,
                                             rows));
  drawer.setOnItemClickListener(this);

  drawerLayout=(DrawerLayout)findViewById(R.id.drawer_layout);
  toggle=
      new ActionBarDrawerToggle(this, drawerLayout,
                                R.drawable.ic_drawer,
                                R.string.drawer_open,
                                R.string.drawer_close);
  drawerLayout.setDrawerListener(toggle);
  getActionBar().setDisplayHomeAsUpEnabled(true);
  getActionBar().setHomeButtonEnabled(true);

  getFragmentManager().addOnBackStackChangedListener(this);

  if (getFragmentManager().findFragmentById(R.id.content) == null) {
    showLorem();
  }
}
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/MainActivity.java)

Our `showLorem()` method now will `post()` a `Runnable`, named `onNavChange`, after it calls `commit()` on its `FragmentTransaction`:

```
private void showLorem() {
  if (lorem == null) {
    lorem=new LoremFragment();
  }

  if (!lorem.isVisible()) {
    getFragmentManager().popBackStack();
    getFragmentManager().beginTransaction()
                        .replace(R.id.content, lorem).commit();
    drawer.post(onNavChange);
  }
}
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/MainActivity.java)

That magic `onNavChange` `Runnable` simply sees what fragment is presently visible and updates the checked item in the nav drawer's `ListView` to match:

```
private Runnable onNavChange=new Runnable() {
  @Override
  public void run() {
    if (lorem != null && lorem.isVisible()) {
      drawer.setItemChecked(0, true);
    }
    else if (content != null && content.isVisible()) {
      drawer.setItemChecked(1, true);
    }
    else {
      int toClear=drawer.getCheckedItemPosition();

      if (toClear >= 0) {
        drawer.setItemChecked(toClear, false);
      }
    }
  }
};
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/MainActivity.java)

By using `post()`, we schedule the `Runnable` to be executed after the fragment change has occurred. Ideally, we would somehow more explicitly attach the `Runnable` to the `FragmentTransaction`, but that does not appear to be an option.

Since we call `showLorem()` in `onCreate()`, this causes our navigation drawer to show that we are in `LoremFragment` when the activity first starts up:

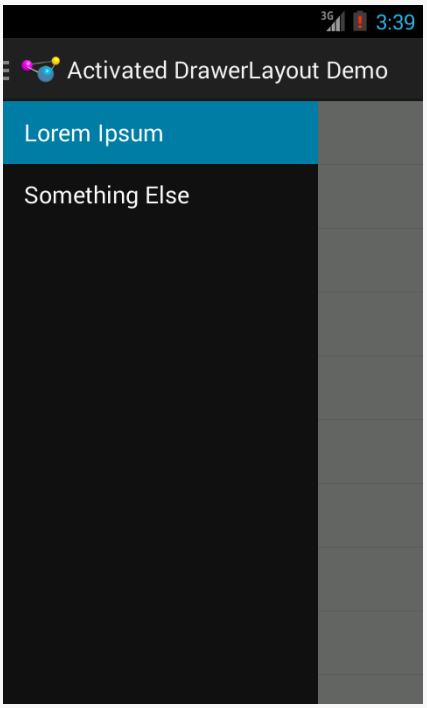


Figure 532: Activated Navigation Drawer Demo, As Initially Launched

The natural behavior of a single-choice `ListView` will check the row that we tap upon, and so when the user taps on an entry in the navigation drawer, it automatically becomes activated. This means we do not have to do our own work for that.

If *everything* in your app is represented by an entry in the navigation drawer, your work is done. However, most likely, there are parts of your app that do not directly map to entries in the navigation drawer... and that is where things get a wee bit complex.

To demonstrate this, we need a bit more to our UI. So, we make `LoremFragment` use the contract pattern, seen elsewhere in the book. We override `onListItemClick()` to call a `wordClicked()` method on the contract, to let the hosting activity know about that UI operation:

```
package com.commonsware.android.drawer.activated;

import android.os.Bundle;
```

```
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class LoremFragment extends ContractListFragment<LoremFragment.Contract> {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    setListAdapter(new ArrayAdapter<String>(
                                          getActivity(),
                                          android.R.layout.simple_list_item_1,
                                          items));
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    getContract().wordClicked();
  }

  interface Contract {
    void wordClicked();
  }
}
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/LoremFragment.java)

We also create a `StuffFragment` that displays a simple message:

Figure 533: Activated Navigation Drawer Demo, Showing StuffFragment

The idea is that clicking on a word in the `LoremFragment` should bring up the `StuffFragment`, as if this were a master/detail implementation.

The implementation of `wordClicked()` in `MainActivity` does indeed show a `StuffFragment` (while also adding it to the back stack), but it also needs to make the drawer `ListView` have no checked items, reflecting the fact that the UI state does not reflect one of the navigation destinations:

```
@Override
public void wordClicked() {
  if (stuff == null) {
    stuff=new StuffFragment();
  }

  getFragmentManager().beginTransaction()
                      .replace(R.id.content, stuff)
                      .addToBackStack(null).commit();
  drawer.post(onNavChange);
}
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/MainActivity.java)

Note that we also `post()` the `onNavChange` `Runnable` here as well.

The result is that when we do tap on a word in the list, the `StuffFragment` appears, but the navigation drawer shows no activated row:

Figure 534: Activated Navigation Drawer Demo, Showing StuffFragment and Navigation Drawer

However, since we manually cleared the checked state, we now need to re-check a row if the user navigates back to one of the other fragments. There are two ways the user could return to one of those other fragments: via the navigation drawer, or via the BACK button (popping our transaction off the back stack).

If they navigate to one of the other fragments via the navigation drawer, the appropriate row will be activated automatically by the user's click event. However, we need to consider what to do about that transaction hanging around the back stack from before. One option is to remove it, so the other fragments behave as they do normally, where BACK exits the activity. That is a matter of calling `popBackStack()` on the `FragmentManager` as part of showing one of the fragments, such as the `showContent()` method that shows the `ContentFragment`:

```
private void showContent() {
  if (content == null) {
    content=new ContentFragment();
  }

  if (!content.isVisible()) {
    getFragmentManager().popBackStack();
    getFragmentManager().beginTransaction()
                        .replace(R.id.content, content).commit();
    drawer.post(onNavChange);
  }
}
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/MainActivity.java)

If the user presses the BACK button, having the earlier fragment reappear happens automatically. However, we need to fix up the navigation drawer to show the proper row as being activated. To do that, we implement `OnBackStackChangedListener` on `MainActivity` and call `addOnBackStackChangedListener()` on the `FragmentManager` in the `onCreate()` initialization work. That way, `onBackStackChanged()` will be called when there is a change in the state of the back stack. Then, it is merely a matter of calling `post()` for `onNavChanged` again, to update the nav drawer:

```
@Override
public void onBackStackChanged() {
  drawer.post(onNavChange);
}
```

(from NavDrawer/Activated/app/src/main/java/com/commonsware/android/drawer/activated/MainActivity.java)

Hiding Context-Specific Action Bar Items

Another Google design guideline that makes sense, but adds complexity, is to only show action bar items that pertain to the entire application while the navigation drawer is visible. So, for example, a "Help" action bar item should remain visible, to allow users to switch over to that. But an "Edit" action bar item, to edit something in the main activity, should be hidden while the navigation drawer is visible.

The navigation drawer is effectively an application-level construct, even if we wind up implementing it on a per-activity basis due to the way Android user interfaces are constructed. Hence, the action bar items with the drawer open should pertain to the same scope that the drawer itself does: the application, not a particular activity or fragment inside of it.

Also, on phone-sized screens, the user may not be able to see much of the underlying UI, as the drawer itself will occlude most of it. They may not remember exactly what was showing, and therefore may forget what that "Delete" action bar item would actually delete. Hiding such a context-specific item, while the drawer is open, is safer.

The converse, of course, is that when the drawer closes, you will need to show once again the items that you hid when the drawer opened.

`DrawerLayout` supports a `DrawerListener`, an instance of which you can attach to the drawer itself, to be notified of when the drawer opens and closes, so you can adjust your action bar items to match.

Interacting with an Action Mode

You may be using your own custom action modes (a.k.a., contextual action bars), such as allowing the user to perform operations on multiple selections in a `ListView`.

Since the navigation drawer is tied to the application as a whole, not necessarily just the current activity or the selections in it, Google's recommendation is for you to temporarily dismiss the action mode when the navigation drawer is opened. This will allow the action bar to show items of relevance to the app, not to the selection. If the user navigates elsewhere using the navigation drawer, you would leave the action mode dismissed. If the user slides the navigation drawer closed, though, you could re-enable the action mode, tied back into the multiple selections the user has already made.

Advertising Your Drawer

Users who have spent some time with Android will have a decent shot at recognizing that tapping the action bar item adorned with the nav drawer artwork will open a drawer. However, not all users will necessarily make that connection, particularly users relatively new to Android.

As noted earlier in this chapter, `DrawerLayout` implements the "peek" pattern, where a long-tap on the edge of the screen will cause the drawer to open just a bit, to hint that there is something that can be opened with a swipe gesture. This is nice and subtle, but perhaps too subtle, as users are not necessarily likely to tap-and-hold on the screen edge just to see if something interesting happens.

Another possibility is to have the drawer open automatically, either the first time that your app is launched, or every time your app is launched until you detect that the user has manually opened the drawer (e.g., by adding a `DrawerListener` and watching for `onDrawerSlide()` and `onDrawerOpened()` events not triggered by you). On the plus side, this puts the drawer "front and center", so the user cannot miss it. Once the user taps in your activity (inside or outside of the drawer), the drawer will close with the animated slide to the edge. The hope is that the user will either eventually realize that they can bezel-swipe to get that drawer open, or they will learn about the action bar option (where you have it). This works if the drawer is fairly useful, relative to what the activity would be displaying by default. If, however, the drawer pales in comparison to the main activity view, forcing the drawer open may reduce, not improve, usability.

What Should Not Be in the Drawer

Your navigation drawer should only provide access to the major areas of content in your app. If you do not have many "major areas of content in your app", reconsider having a navigation drawer, and use something else (e.g., `ViewPager` with a tabbed indicator) to get to what content you have.

Also, there are certain things that should not go into a navigation drawer:

- Actions (verbs) generally do not belong in the navigation drawer. "Edit", "Save", and so forth belong as action bar items, or perhaps as part of an action mode, not in the navigation drawer.
- Content itself does not belong in the navigation drawer. Bits of metadata, in the form of badges and similar sorts of indicators, are fine, to hint to the user about the content that is available in that area of your app. But the navigation drawer is not meant to be the "master" in the master/detail pattern, nor is it some sort of "sidebar" of additional content that you just want to have hidden away.
- Low-priority areas of your app, particularly those that are traditionally in the action bar, should not be in the navigation drawer. "Help" and "About" are classic examples. "Settings" is another, though the line starts to get a bit fuzzy (while "Settings" might be something traditional for the action bar, "Accounts" is not). The overflow menu of an action bar is a fine place to have those sorts of areas be available to the user without cluttering up the primary action bar or the navigation drawer.

Independent Implementations

The navigation drawer pattern did not begin with the introduction of `DrawerLayout`. There have been many independent implementations of such a "sliding menu" that pre-dated `DrawerLayout` and the Android design guidelines for navigation drawers. Examples include:

- Jeremy Feinstein's SlidingMenu
- 6Wunderkinder's Sliding Layer
- David Scott's RibbonMenu

Just bear in mind that these implementations do not necessarily adhere to everything in the design guidelines, requiring you to perhaps make modifications or simply ignore those guidelines as needed.

RecyclerView

Visually representing collections of items is an important aspect of many mobile apps. The classic Android implementation of this was the `AdapterView` family of widgets: `ListView`, `GridView`, `Spinner`, and so on. However, they had their limitations, particularly with respect to advanced capabilities like animating changes in the list contents.

In 2014, Google released `RecyclerView`, via the Android Support package. Developers can add the `recyclerview-v7` to their projects and use `RecyclerView` as a replacement for most of the `AdapterView` family. `RecyclerView` was written from the ground up to be a more flexible container, with lots of hooks and delegation to allow behaviors to be plugged in.

This had two major impacts:

1. `RecyclerView` is indeed much more powerful than its `AdapterView` counterparts
2. `RecyclerView`, out of the box, is nearly useless, and wiring together enough stuff to even replicate basic `ListView`/`GridView` functionality takes quite a bit of code

In this chapter, we will review `RecyclerView` from the ground up, starting with basic operation. Many of the `ListView` samples from elsewhere in the book will be replicated here, to see how to pull off the same things with `RecyclerView`. And, we will explore some of the additional capabilities that make `RecyclerView` perhaps worth the effort on high-end Android applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on AdapterView and adapters.

One section involves the use of custom XML drawables. Another section demonstrates using content pulled from the MediaStore ContentProvider.

This chapter also covers things like action modes and other advanced ListView techniques.

AdapterView and its Discontents

AdapterView, and particularly its ListView and GridView subclasses, serve important roles in Android application development. And, for basic scenarios, they work reasonably well.

However, there are issues.

Perhaps the biggest tactical issue is that updating an AdapterView tends to be an all-or-nothing affair. If there is a change to the model data — new rows added, existing rows removed, or data changes that might affect the AdapterView presentation — the only well-supported solution is to call notifyDataSetChanged() and have the AdapterView rebuild itself. This is slow and can have impacts on things like choice states. And, if you wanted to get really elaborate about your changes, and use animated effects to show where rows got added or removed, that was halfway to impossible.

Strategically, AdapterView, AbsListView (the immediate parent of ListView and GridView), and ListView are large piles of code that resemble pasta to many outsiders. There are so many responsibilities piled into these classes that maintainability was a challenge for Google and extensibility was a dream more than a reality.

Enter RecyclerView

RecyclerView is designed to correct those sorts of flaws.

`RecyclerView`, on its own, does very little other than help manage view recycling (e.g., row recycling of a vertical list). It delegates almost everything else to other classes, such as:

- a layout manager, responsible for organizing the views into various structures (vertical list, grid, staggered grid, etc.)
- an item decorator, responsible for applying effects and light positioning to the views, such as adding divider lines between rows in a vertical list
- an item animator, responsible for animated effects as the model data changes

This is on top of the adapters and view holders that were the hallmarks of conventional `AdapterView` usage.

Because things like layout managers are handled via abstract classes and replaceable concrete implementations, third-party developers can contribute options for developers to use, just as Google does. Later in this chapter, we will explore some of these contributions.

On the flip side, though, `RecyclerView` does *much* less "out of the box" than does `ListView` or `GridView`. Not everything that is missing is supplied anywhere in the `recyclerview-v7` library, requiring that you either roll a bunch of code yourself or rely upon those third-party libraries to get anything much done.

A Trivial List

Back in the original chapter on `AdapterView` and adapters, we had the `Selection/Dynamic` sample app. This app would display a list of 25 Latin words, each with the word's length and an accompanying icon (different for short and long words):

Figure 535: The Dynamic Sample Application

Here, we will review the `RecyclerView/SimpleList` sample project, which is a first pass at porting the `Selection/Dynamic` demo over to use `RecyclerView`.

The Dependency

Any project that wishes to use `RecyclerView` needs to have access to the `recyclerview-v7` library from the Android Support package. Android Studio users can simply have a reference to it in the top-level `dependencies` closure:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.android.support:recyclerview-v7:22.2.0'
}

android {
    compileSdkVersion 22
    buildToolsVersion '26.0.2'
}
```

(from RecyclerView/SimpleList/app/build.gradle)

However, if you are using `recyclerview-v7`, you want to use version 23 or higher of that library. There are changes to ART – the Android runtime used on Android 5.0+ — that apparently will break the older versions of `recyclerview-v7` when running on Android 6.0+ devices.

A RecyclerViewActivity

With `ListView`, we could use `ListActivity`, to isolate some of the `ListView`-management code. There is no `RecyclerViewActivity` in the `recyclerview-v7` library... but we can create one:

```
package com.commonsware.android.recyclerview.simplelist;

import android.app.Activity;
import android.support.v7.widget.RecyclerView;

public class RecyclerViewActivity extends Activity {
  private RecyclerView rv=null;

  public void setAdapter(RecyclerView.Adapter adapter) {
    getRecyclerView().setAdapter(adapter);
  }

  public RecyclerView.Adapter getAdapter() {
    return(getRecyclerView().getAdapter());
  }

  public void setLayoutManager(RecyclerView.LayoutManager mgr) {
    getRecyclerView().setLayoutManager(mgr);
  }

  public RecyclerView getRecyclerView() {
    if (rv==null) {
      rv=new RecyclerView(this);
      rv.setHasFixedSize(true);
      setContentView(rv);
    }

    return(rv);
  }
}
```

(from RecyclerView/SimpleList/app/src/main/java/com/commonsware/android/recyclerview/simplelist/RecyclerViewActivity.java)

The important part is the `getRecyclerView()` method. Here, if we have not already initialized the `RecyclerView`, we create an instance of it and set it as the activity's

content view via `setContentView()`. Along the way, we call `setHasFixedSize(true)` on the `RecyclerView`, to tell it that its size should not be changing based upon the contents of the adapter. This knowledge can help `RecyclerView` operate more efficiently.

The `RecyclerViewActivity` also has `getAdapter()` and `setAdapter()` analogues for their `ListActivity` counterparts. We will explore the differences in the adapter classes later in this section. We also have a `setLayoutManager()` convenience method, that just calls `setLayoutManager()` on the underlying `RecyclerView` — we will see what a layout manager is in the context of `RecyclerView` in the next section.

There are other features of `ListActivity` that are not mirrored here in `RecyclerViewActivity`, just to keep `RecyclerViewActivity` short. Notably, `ListActivity` supports either inflating a custom layout that contains the `ListView` *or* creating its own. `RecyclerViewActivity` does not support this, though it could with some minor extensions.

The LayoutManager

The "real" activity of the project is `MainActivity`, which consists of a single method: `onCreate()`

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  setLayoutManager(new LinearLayoutManager(this));
  setAdapter(new IconicAdapter());
}
```

(from RecyclerView/SimpleList/app/src/main/java/com/commonsware/android/recyclerview/simplelist/MainActivity.java)

After chaining to the superclass, the first thing we do is call `setLayoutManager()`, which will associate a `RecyclerView.LayoutManager` with our `RecyclerView`. Specifically, we are using a `LinearLayoutManager`.

`ListView` has the notion of a vertically-scrolling list of rows "baked into" its implementation. Similarly, `GridView` has the notion of a two-dimensional vertically-scrolling grid "baked into" its implementation. `RecyclerView`, on the other hand, knows absolutely nothing about how to lay out its children. That work is delegated to a `RecyclerView.LayoutManager`, so that different approaches can be plugged in as needed.

There are three concrete subclasses of the abstract `RecyclerView.LayoutManager` base class that ship with `recyclerview-v7`:

- `LinearLayoutManager`, which implements a vertically-scrolling list, akin to `ListView`
- `GridLayoutManager`, which implements a two-dimensional vertically-scrolling list, akin to `GridView`
- `StaggeredGridLayoutManager`, which implements a "staggered grid", which has columns of cells like a `GridView`, but where the cells do not have to all have the same size

In addition, it is eminently possible to create your own `RecyclerView.LayoutManager`, or use ones from third-party libraries.

In this example, though, we stick with a simple `LinearLayoutManager`, as we are attempting to replicate the functionality of a `ListView`.

The Adapter

Our `onCreate()` method also calls `setAdapter()`, to associate an `RecyclerView.Adapter` with our `RecyclerView` (specifically, a revised version of our `IconicAdapter` from the original `Selection/Dynamic` sample app). As with the `AdapterView` family, `RecyclerView` uses an adapter to help convert our model data into visual representations. However, the implementation of a `RecyclerView.Adapter` is substantially different from a classic `ListAdapter` for use with `ListView` or `GridView`.

Reminiscent of `ArrayAdapter`, a `RecyclerView.Adapter` uses generics, and we declare what sort of stuff we are adapting. However, `ArrayAdapter` uses the generic to describe the model data. `RecyclerView.Adapter` instead uses the generic to identify a `ViewHolder` that will be responsible for doing the work to actually tie model data to row widgets:

```
class IconicAdapter extends RecyclerView.Adapter<RowHolder> {
  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowHolder(getLayoutInflater()
                         .inflate(R.layout.row, parent, false)));
  }

  @Override
  public void onBindViewHolder(RowHolder holder, int position) {
```

```
    holder.bindModel(items[position]);
  }

  @Override
  public int getItemCount() {
    return(items.length);
  }
}
```

(from RecyclerView/SimpleList/app/src/main/java/com/commonsware/android/recyclerview/simplelist/MainActivity.java)

In our case, `IconicAdapter` is using a `RowHolder` class that we will examine in the next section.

A `RecyclerView.Adapter` has three abstract methods that need to be implemented.

One is `getItemCount()`, which fills the same role as does `getCount()` with a `ListAdapter`, indicating how many items there will be in the `RecyclerView`. In the case of `IconicAdapter`, this is based on the length of the `items` `static` array of `String` objects, same as it was with `IconicAdapter` in the `Selection/Dynamic` sample app:

```
private static final String[] items={"lorem", "ipsum", "dolor",
        "sit", "amet",
        "consectetuer", "adipiscing", "elit", "morbi", "vel",
        "ligula", "vitae", "arcu", "aliquet", "mollis",
        "etiam", "vel", "erat", "placerat", "ante",
        "porttitor", "sodales", "pellentesque", "augue", "purus"};
```

(from RecyclerView/SimpleList/app/src/main/java/com/commonsware/android/recyclerview/simplelist/MainActivity.java)

The other two methods are `onCreateViewHolder()` and `onBindViewHolder()`. These are a bit reminiscent of the `newView()` and `bindView()` methods that are used by a `CursorAdapter`. However, rather than working directly with views, `onCreateViewHolder()` and `onBindViewHolder()` work with `ViewHolder` objects, as a formalization of the view holder pattern seen originally in the chapter on selection widgets.

`onCreateViewHolder()`, as the name suggests, needs to create, configure, and return a `ViewHolder` for a particular row of our list. It is passed two parameters:

- a `ViewGroup` that will hold the views managed by the holder, mostly for use with layout inflation, and

- an `int` that is the particular view type we are using, for cases where we have multiple view types

The `IconicAdapter` implementation inflates our row view (`R.layout.row`) and passes it to the `RowHolder` constructor, returning the resulting `RowHolder`.

`onBindViewHolder()` is responsible for updating a `ViewHolder` based upon the model data for a certain `position`. `IconicAdapter` handles this by passing the model into a private `bindModel()` method implemented on `RowHolder`.

There are many other methods you could override on `RecyclerView.Adapter`, and we will see a few of those later in this chapter. But, for a simple list, these three will suffice.

The ViewHolder

The `RecyclerView.ViewHolder` is responsible for binding data as needed from our model into the widgets for a row in our list:

```
static class RowHolder extends RecyclerView.ViewHolder {
  TextView label=null;
  TextView size=null;
  ImageView icon=null;
  String template=null;

  RowHolder(View row) {
    super(row);

    label=(TextView)row.findViewById(R.id.label);
    size=(TextView)row.findViewById(R.id.size);
    icon=(ImageView)row.findViewById(R.id.icon);

    template=size.getContext().getString(R.string.size_template);
  }

  void bindModel(String item) {
    label.setText(item);
    size.setText(String.format(template, item.length()));

    if (item.length()>4) {
      icon.setImageResource(R.drawable.delete);
    }
    else {
      icon.setImageResource(R.drawable.ok);
    }
```

```
    }
}
```

(from RecyclerView/SimpleList/app/src/main/java/com/commonsware/android/recyclerview/simplelist/MainActivity.java)

However, other than needing to use the base class of `RecyclerView.ViewHolder`, there is no other particular protocol that is mandated between the adapter and the view holder. You can invent your own API. Here, we use the `RowHolder` constructor to pass in the row `View`, where the constructor retrieves the individual widgets and sets up our string resource template. Then, a private `bindModel()` method takes our model object (a `String`) and binds it to the row's widgets, applying our business rules along the way.

The Results

As the project name suggests, this gives us a simple list:

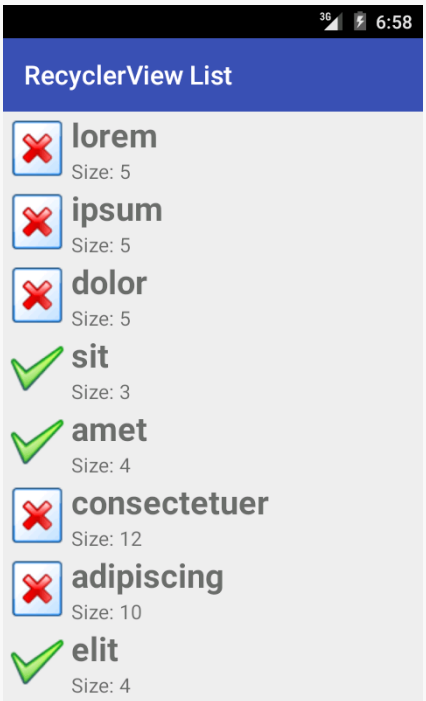


Figure 536: SimpleList RecyclerView Demo

As with `ListView`, `RecyclerView` (along with the `RecyclerView.LayoutManager`) handles the vertical scrolling through our available rows.

What's Missing?

However, we are lacking two things that we had in the `Selection/Dynamic` edition of this sample that used a `ListView`.

First, there are no dividers between the rows. That may not be a huge issue for this particular row layout, but other layouts may need more assistance in visually separating one row from the next. We will explore ways of accomplishing this in the next section.

Second, we are missing click events. The user can tap on rows as much as she wants. Not only will the user not get any visual feedback from those taps, but we have no `setOnItemClickListener()` to find out about those taps. We will explore how to fill in this gap later in the chapter.

`RecyclerView` also lacks a variety of other things that we could get from a `ListView`, that we happen to not be using in this sample, such as:

- choice modes, for checklists and such
- header and footer views
- any concept of a "selected" row
- filter support
- and so on

We will explore some of those and how to address them in this chapter.

Divider Options

There are two main approaches for visually separating items in a `RecyclerView`:

1. Ensure that this is handled via the layout itself, such as using a `CardView`
2. Use a `RecyclerView.ItemDecoration` to apply a common divider between items

Both of these techniques will be covered in this chapter.

CardView

Cards are a popular visual metaphor in mobile development. Dividing content collections (or aspects of a larger piece of content) into cards makes it clearer how

you can reorganize that content to fit various screen sizes and orientations. In some cases, you might have a single column of cards, while in other cases, you have cards arranged more laterally.

In 2014, Google released `cardview-v7`, another library in the Android Support package, that offers a `CardView`. `CardView` is a simple subclass of `FrameLayout`, designed to provide a card UI, consisting of a rounded rectangle and a drop shadow. In particular, `CardView` will use Android 5.0's default drop shadows based on widget elevation, while offering emulated drop shadows on earlier Android releases. This way, you can get a reasonably consistent look going back to API Level 7.

To use this, you will have to add the `cardview-v7` library to your app project. Android Studio users can just add a dependency on the `cardview-v7` artifact in the Android Support repository, as seen in the `RecyclerView/CardViewList` sample project:

```
dependencies {
    implementation 'com.android.support:recyclerview-v7:22.2.0'
    implementation 'com.android.support:cardview-v7:22.2.0'
}
```

(from RecyclerView/CardViewList/app/build.gradle)

Then, you can wrap your row layout in a `CardView` (or, more accurately, in an `android.support.v7.widget.CardView`):

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:cardview="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="4dp"
  cardview:cardCornerRadius="4dp">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
```

```
      android:padding="2dip"
      android:src="@drawable/ok"
      android:contentDescription="@string/icon"/>

    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="25sp"
        android:textStyle="bold"/>

      <TextView
        android:id="@+id/size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="15sp"/>
    </LinearLayout>

  </LinearLayout>
</android.support.v7.widget.CardView>
```

(from RecyclerView/CardViewList/app/src/main/res/layout/row.xml)

With no other code changes from the original `RecyclerView/SimpleList` sample, we get this:

Figure 537: CardViewList RecyclerView Demo

Note that drop shadows from `CardView` may not show up on Android 5.0+ emulators, particularly if you have Host GPU mode disabled in the emulator AVD. The `CardView` itself will work fine, just without the drop shadow effect.

Manual

A `CardView` may not be an appropriate visual approach for your list. Perhaps you want a regular divider, like we had with `ListView`. While that is possible, it is not especially straightforward.

`RecyclerView` considers things like dividers to be "item decorations". There is a `RecyclerView.ItemDecoration` abstract class that you can extend to handle item decoration, and you can attach such a decoration to a `RecyclerView` via `addItemDecoration()`. As the name suggests, you can have more than one decorator if needed.

Originally, Google did not bother to provide any concrete implementation of such a decoration. However, they eventually added a `DividerItemDecoration` class that you can use for a simple, "out of the box" divider.

DividerItemDecoration

The RecyclerView/DividerList sample project demonstrates the use of `DividerItemDecoration`... which consists of a single call to `addItemDecoration()`:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  setLayoutManager(new LinearLayoutManager(this));

  getRecyclerView()
    .addItemDecoration(new DividerItemDecoration(this, LinearLayoutManager.VERTICAL));
  setAdapter(new IconicAdapter());
}
```

(from RecyclerView/DividerList/app/src/main/java/com/commonsware/android/recyclerview/divider/MainActivity.java)

The `DividerItemDecoration` constructor just takes a `Context` and the orientation to use. Here, though, the orientation refers to *the orientation of the `RecyclerView`*. You might think that using `LinearLayoutManager.VERTICAL` here would mean that the divider would be drawn vertically. Instead, it means that the `RecyclerView` scrolls vertically, so dividers are drawn horizontally.

While `DividerItemDecoration` has a few configuration options — notably, a `setDrawable()` method to override the default artwork to use for the divider – the "out of the box" implementation largely matches the look of the `ListView` divider:

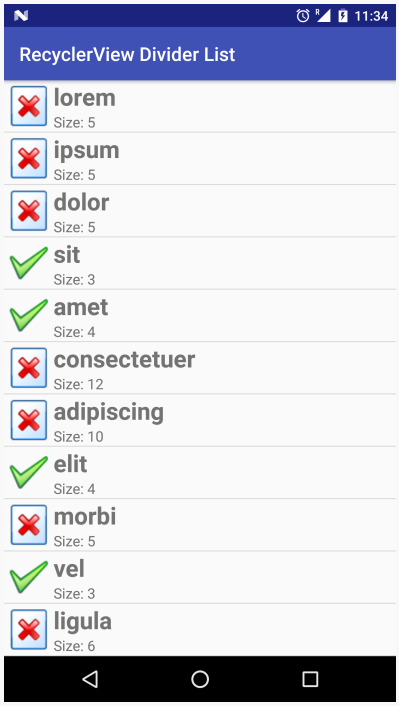


Figure 538: DividerItemDecoration, Applied to a RecyclerView

DIY Decorators

As noted above, originally `recyclerview-v7` did not come with an actual divider. A few enterprising developers experimented with this, leading to solutions like this one, published as a GitHub gist. The `RecyclerView/ManualDividerList` sample project demonstrates the use of such a decoration. It may be that what *you* want for a decorator is more complex than what `DividerItemDecoration` can offer, and so implementing your own decorator may be necessary.

First, we will need a drawable resource for the divider itself:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
       android:shape="rectangle">

  <size
    android:width="1dp"
    android:height="1dp" />

  <solid android:color="@color/divider" />
```

```
</shape>
```

(from RecyclerView/ManualDividerList/app/src/main/res/drawable/item_divider.xml)

This is a `ShapeDrawable`, as is covered in the chapter on drawables. The big thing is the `solid` fill, here pointing to a color resource for the color to use for that fill:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="divider">#ffaaaaaa</color>
</resources>
```

(from RecyclerView/ManualDividerList/app/src/main/res/values/colors.xml)

The `ShapeDrawable` is given a size of 1dp square. In reality, it will be resized on the fly by the decorator to fill the width of the `RecyclerView`.

Note that there is nothing especially magic about using this particular drawable. You could have a gradient fill to have the divider taper off towards the ends and be solid in the middle. Or, you could use a nine-patch PNG file, a `VectorDrawable` on Android 5.0+, or anything else that will resize well.

Next, we need a `RecyclerView.ItemDecoration` implementation, such as the sample project's `HorizontalDividerItemDecoration`:

```
package com.commonsware.android.recyclerview.manualdivider;

import android.graphics.Canvas;
import android.graphics.drawable.Drawable;
import android.support.v7.widget.RecyclerView;
import android.view.View;

// inspired by https://gist.github.com/polbins/e37206fbc444207c0e92

public class HorizontalDividerItemDecoration extends RecyclerView.ItemDecoration {
  private Drawable divider;

  public HorizontalDividerItemDecoration(Drawable divider) {
    this.divider=divider.mutate();
  }

  @Override
  public void onDrawOver(Canvas c, RecyclerView parent, RecyclerView.State state) {
    int left=parent.getPaddingLeft();
    int right=parent.getWidth()-parent.getPaddingRight();

    int childCount=parent.getChildCount();

    for (int i=0; i<childCount-1; i++) {
      View child=parent.getChildAt(i);
```

```
      RecyclerView.LayoutParams params=
          (RecyclerView.LayoutParams)child.getLayoutParams();

      int top=child.getBottom()+params.bottomMargin;
      int bottom=top+divider.getIntrinsicHeight();

      divider.setBounds(left, top, right, bottom);
      divider.draw(c);
    }
  }
}
```

(from RecyclerView/ManualDividerList/app/src/main/java/com/commonsware/android/recyclerview/manualdivider/HorizontalDividerItemDecoration.java)

This class takes the `Drawable` that is the divider as input, so it can be used for different dividers as needed. `HorizontalDividerItemDecoration` calls `mutate()` on the `Drawable` to get a `Drawable` that can be changed independently of any original instance of the `Drawable`. This is important when using `Drawable` resources, as the `Drawable` instances get reused for other references to the same resource, so changing the core `Drawable` itself (e.g., via a `setBounds()` call) is unsafe.

The main logic of `HorizontalDividerItemDecoration` resides in the `onDrawOver()` method. This will be called to let us draw over top of the items in the `RecyclerView`. Here we:

- Determine the left and right extents to draw, relative to the left and right edges of the `RecyclerView`, but subtracting the padding, so that we only draw inside of that padding
- Iterate over the child views of the `RecyclerView`, find the vertical location for that divider, resize the divider to fit the desired space, and then draw the divider on the supplied `Canvas`, skipping the last child so we do not draw a divider at the bottom of the list

Using that bit of magic, then, is merely a matter of attaching our `HorizontalDividerItemDecoration` to our `RecyclerView`, done here in `onCreate()` of `MainActivity`:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  setLayoutManager(new LinearLayoutManager(this));

  Drawable divider=getResources().getDrawable(R.drawable.item_divider);

  getRecyclerView().addItemDecoration(new HorizontalDividerItemDecoration(divider));
  setAdapter(new IconicAdapter());
}
```

(from RecyclerView/ManualDividerList/app/src/main/java/com/commonsware/android/recyclerview/manualdivider/MainActivity.java)

The rest of the sample project is a clone of the original `SimpleList` sample project from the beginning of this chapter.

The result is that we have a divider drawn between the children:

Figure 539: ManualDividerList RecyclerView Demo

Handling Click Events

However, having nice dividers does not address the larger problem: responding to input.

The `RecyclerView` vision, overall, is that `RecyclerView` itself has nothing much to do with input, other than scrolling. Anything having to do with users clicking things and triggering some sort of response is the responsibility of the views inside the `RecyclerView`, such as the rows in a list-style `RecyclerView`.

This has its benefits. Clickable widgets, like a `RatingBar`, in a `ListView` row had long been in conflict with click events on rows themselves. Getting rows that can be clicked, with row contents that can also be clicked, gets a bit tricky at times. With

RecyclerView, you are in more explicit control over how this sort of thing gets handled... because you are the one setting up all of the on-click handling logic.

Of course, that does not help the users much. Users do not care what bit of code is responsible for input. Users simply want to provide the input. If you present them with a vertically-scrolling list-style UI, they will attempt to click on rows in the list and will expect some sort of outcome.

The RecyclerView approach, though, means that you are largely on your own for handling that input. This requires yet more code that, in an ideal world, would be offered as an "out of the box" option by RecyclerView.

Responding to Clicks

At its core, responding to clicks is a matter of setting an OnClickListener on the appropriate Views.

So, for example, the RecyclerView/CardClickList sample project is a clone of the CardViewList sample, where we call setOnClickListener() on the row View in the RecyclerView.ViewHolder, now renamed RowController:

```
package com.commonsware.android.recyclerview.cardclicklist;

import android.support.v7.widget.RecyclerView;
import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;

class RowController extends RecyclerView.ViewHolder
    implements View.OnClickListener {
  TextView label=null;
  TextView size=null;
  ImageView icon=null;
  String template=null;

  RowController(View row) {
    super(row);

    label=(TextView)row.findViewById(R.id.label);
    size=(TextView)row.findViewById(R.id.size);
    icon=(ImageView)row.findViewById(R.id.icon);

    template=size.getContext().getString(R.string.size_template);
```

```
    row.setOnClickListener(this);
  }

  @Override
  public void onClick(View v) {
    Toast.makeText(v.getContext(),
        String.format("Clicked on position %d", getAdapterPosition()),
        Toast.LENGTH_SHORT).show();
  }

  void bindModel(String item) {
    label.setText(item);
    size.setText(String.format(template, item.length()));

    if (item.length()>4) {
      icon.setImageResource(R.drawable.delete);
    }
    else {
      icon.setImageResource(R.drawable.ok);
    }
  }
}
```

(from RecyclerView/CardClickList/app/src/main/java/com/commonsware/android/recyclerview/cardclicklist/RowController.java)

In this sample, all the `onClick()` method does is show a `Toast`. However, you could:

- Raise an event on an event bus, or
- Call a method on some supplied interface (e.g., passed into the `RowController` constructor) to delegate the event to a higher-order controller, or
- Whatever else might be needed

In this case, since none of the widgets in the row are interactive and might consume click events themselves, the user can tap anywhere on the row, and the `Toast` will appear. If you have more complex scenarios — such as a checklist where you have a `CheckBox` in the rows — you can decide for yourself how to handle click events on different parts of the row. We will see checklists in action later in this chapter.

Visual Impact of Clicks

However, if you run the `CardClickList` sample, you will notice one major remaining flaw: there is no visual feedback to the user about the click event. Yes, the `Toast` appears, but users are used to seeing some sort of transient state change in the row

itself on a click, such as a flash of color. Once again, we have the ability to control this as we see fit... by having the responsibility to make it happen at all.

There are a few approaches to this problem, such as the ones outlined in this section.

Option #1: Translucent Selector on Top

An approach that Mark Allison suggested in his Styling Android blog mimics the `drawSelectorOnTop` approach available to `ListView`. Using something like a `FrameLayout`, you layer a translucent selector atop the rows, where the selector implements the click feedback.

The RecyclerView/CardRippleList sample project is a clone of `CardClickList` that takes Mr. Allison's approach. The revised `row.xml` takes advantage of the fact that `CardView` is a subclass of `FrameLayout`, so it layers a plain `View` atop the `LinearLayout` that is the core content of the row:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:cardview="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="4dp"
  cardview:cardCornerRadius="4dp">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:padding="2dip"
      android:src="@drawable/ok"
      android:contentDescription="@string/icon"/>

    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
```

```
      <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="25sp"
        android:textStyle="bold"/>

      <TextView
        android:id="@+id/size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="15sp"/>
    </LinearLayout>

  </LinearLayout>

  <View
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="?android:attr/selectableItemBackground" />

</android.support.v7.widget.CardView>
```

(from RecyclerView/CardRippleList/app/src/main/res/layout/row.xml)

The `background` of that `View` is the `selectableItemBackground` from the current theme. On apps using `Theme`, this will give you an orange flash. On apps using `Theme.Holo`, this will give you a blue flash. On apps using `Theme.Material`, this will give you a ripple animation. And, of course, you can supply your own override value for `selectableItemBackground` to use your own `StateListDrawable` instead.

The downsize of this approach is that the `View` is higher on the Z axis than is the rest of the row content. In this case, since the rest of the row content is non-interactive, this is not a problem. However, if we elect to put interactive widgets in the rows — such as `CheckBox` widgets to implement a checklist — now our `View` will prevent the user from interacting with those widgets.

Option #2: Background Selector

Another approach would be to apply the `selectableItemBackground` to our existing row content, rather than to some separate selector widget that overlays the row content. This is the approach taken in the `RecyclerView/CardRippleList2` sample

project. Here, the `selectableItemBackground` is applied to the `LinearLayout` inside of the `CardView`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:cardview="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="4dp"
  cardview:cardCornerRadius="4dp">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:background="?android:attr/selectableItemBackground">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:padding="2dip"
      android:src="@drawable/ok"
      android:contentDescription="@string/icon"/>

    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="25sp"
        android:textStyle="bold"/>

      <TextView
        android:id="@+id/size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="15sp"/>
    </LinearLayout>

  </LinearLayout>
```

```
</android.support.v7.widget.CardView>
```

(from RecyclerView/CardRippleList2/app/src/main/res/layout/row.xml)

For non-interactive widgets, like our `TextViews` and `ImageView`, touch events will get propagated to the `LinearLayout`, which will trigger the changes in the state of the `StateListDrawable` that is the `LinearLayout` background. Yet, if we change the rows to have interactive widgets, those widgets will still be able to process their own touch events, as we will see later in this chapter.

However, particularly for this sample app, the visual effect is largely the same as with `CardRippleList`: the user will get click feedback based upon the `selectableItemBackground` in use given the activity's theme.

Option #3: Controlled Ripple Emanation Point

There is one problem with both click event implementations, though: the ripples on Android 5.0 start in the center of each row. According to the Material Design rules, the ripples should start where the touch event occurs, so they seem to flow outward from the finger. This was addressed in Android 5.1, but was a bug in Android 5.0

To fix this, you need to use the `setHotspot()` method, added to `Drawable` in API Level 21. `setHotspot()` provides to the drawable a "hot spot", and `RippleDrawable` apparently uses this as the emanation point for the ripple effect. `setHotspot()` takes a pair of `float` values, presumably with an eye towards using `setHotspot()` inside of an `OnTouchListener`, as the `MotionEvent` reports X/Y positions of the touch event with `float` values.

The `RecyclerView/CardRippleList3` sample project is a clone of `CardRipple2` that adds this feature.

The row layout is the same as before. However, in `RowController`, when setting up the row, we register an `OnTouchListener`, to find out the low-level `MotionEvent` of when the user touches our row:

```
RowController(View row) {
  super(row);

  label=(TextView)row.findViewById(R.id.label);
  size=(TextView)row.findViewById(R.id.size);
  icon=(ImageView)row.findViewById(R.id.icon);
```

```
    template=size.getContext().getString(R.string.size_template);

    row.setOnClickListener(this);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
      row.setOnTouchListener(new View.OnTouchListener() {
        @TargetApi(Build.VERSION_CODES.LOLLIPOP)
        @Override
        public boolean onTouch(View v, MotionEvent event) {
          v
            .findViewById(R.id.row_content)
            .getBackground()
            .setHotspot(event.getX(), event.getY());

          return(false);
        }
      });
    }
  }
```

(from RecyclerView/CardRippleList3/app/src/main/java/com/commonsware/android/recyclerview/cardripplelist3/RowController.java)

We only bother registering this listener on API Level 21+, as there is no `setHotspot()` method on prior versions of Android and therefore no need for the listener. However, *if* we are on an Android 5.0+ device, we intercept the touch event, pass it along to `setHotspot()` on the background `Drawable`, and return `false` to ensure that regular touch event processing proceeds.

The effect is subtle and may be difficult for you to discern. But, if you look at the touch events in slow motion (e.g., screen record a session, then examine the resulting video frame-by-frame), you will see that the ripple effect appears to emanate from the touch point, rather than from the row's center as before. And, since this logic is only used on API Level 21+, older devices are unaffected.

What About Cursors?

So far, our model data has been a simple static array. Often times, though, we need to be working with model data culled from a database or `ContentProvider`. It may be that, for other reasons, we want to convert the `Cursor` we get back from queries into an array of ordinary Java objects. However, there is nothing stopping us from using a `Cursor` more directly as the model for a `RecyclerView`.

The `RecyclerView.Adapter` is responsible for teaching the `RecyclerView.ViewHolder` the model data to bind against. The

`RecyclerView.Adapter` base class is oblivious to how that model data is organized: array, `ArrayList`, `Cursor`, `JSONArray`, etc. And the actual bind-the-data logic for the `ReyclerView.ViewHolder` is our responsibility — again, the base class is oblivious to where the data is coming from. Hence, we can create our own protocol for passing the model data for the needed `position` from the `RecyclerView.Adapter` to the `RecyclerView.ViewHolder`. If we want to use a `Cursor` as the vehicle for doing this, we are welcome to do so.

This is illustrated in the `RecyclerView/VideoList` sample project, which is a clone of the `VideoList` project introduced in the chapter on the `MediaStore` `ContentProvider`. In the original sample, the list was a `ListView`; in this sample, the list is a `RecyclerView`.

The core "plumbing" of the app is akin to the previous `RecyclerView` samples, such as using `RecyclerViewActivity` for handling getting the `RecyclerView` on the screen. However, our row layout is now based on the original `VideoList` row:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:padding="8dp"
  android:background="?android:attr/selectableItemBackground">

  <ImageView
    android:id="@+id/thumbnail"
    android:layout_width="64dp"
    android:layout_height="64dp"
    android:contentDescription="@string/thumbnail"/>

  <TextView
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_gravity="center_vertical"
    android:textSize="24sp"/>

</LinearLayout>
```

(from RecyclerView/VideoList/app/src/main/res/layout/row.xml)

However, as we will now be accessing media, we need the `READ_EXTERNAL_STORAGE` permission, so we request that in the manifest:

```
<?xml version="1.0"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.commonsware.android.recyclerview.videolist"
          android:versionCode="1"
          android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

  <application
    android:icon="@drawable/ic_launcher"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from RecyclerView/VideoList/app/src/main/AndroidManifest.xml)

And our `app/build.gradle` file gives us a `targetSdkVersion` of 23, requiring us to deal with runtime permissions on Android 6.0+:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.android.support:recyclerview-v7:23.4.0'
    implementation 'com.squareup.picasso:picasso:2.5.2'
}

android {
    compileSdkVersion 23
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
```

```
        targetSdkVersion 23
    }
}
```

(from RecyclerView/VideoList/app/build.gradle)

onCreate() sets up the empty RecyclerView with a LinearLayoutManager and a VideoAdapter (that we will examine shortly). However, we also confirm whether we have READ_EXTERNAL_STORAGE already — if yes, we call loadVideos() to get the videos. If we do not have permission, and we are not in the middle of requesting permission, we ask for permission using requestPermissions():

```
  private static final String STATE_IN_PERMISSION="inPermission";
  private static final int REQUEST_PERMS=137;
  private boolean isInPermission=false;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setLayoutManager(new LinearLayoutManager(this));
    setAdapter(new VideoAdapter());

    if (state!=null) {
      isInPermission=
        state.getBoolean(STATE_IN_PERMISSION, false);
    }

    if (hasFilesPermission()) {
      loadVideos();
    }
    else if (!isInPermission) {
      isInPermission=true;

      ActivityCompat.requestPermissions(this,
        new String[] {Manifest.permission.READ_EXTERNAL_STORAGE},
        REQUEST_PERMS);
    }
  }
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

hasFilesPermission() just uses checkSelfPermission() to see whether we can read external storage:

```
  private boolean hasFilesPermission() {
    return(ContextCompat.checkSelfPermission(this,
```

```
    Manifest.permission.READ_EXTERNAL_STORAGE)==
    PackageManager.PERMISSION_GRANTED);
}
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

We then call `loadVideos()` once we have permission, plus keep track of whether or not we are in the process of requesting permissions (so we do not raise the permission dialog again if we undergo a configuration change while the permission dialog is already on-screen):

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putBoolean(STATE_IN_PERMISSION, isInPermission);
}

@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  isInPermission=false;

  if (requestCode==REQUEST_PERMS) {
    if (hasFilesPermission()) {
      loadVideos();
    }
    else {
      finish(); // denied permission, so we're done
    }
  }
}
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

`loadVideos()` just calls `initLoader()` to request that we load the videos from the `MediaStore`:

```
private void loadVideos() {
  getLoaderManager().initLoader(0, null, this);
}
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

The CursorLoader logic, for getting details about videos from the MediaStore, is pretty much the same as before, other than providing the Cursor to the VideoAdapter when it is ready:

```
@Override
public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
  return(new CursorLoader(this,
                          MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
                          null, null, null,
                          MediaStore.Video.Media.TITLE));
}

@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
  ((VideoAdapter)getAdapter()).setVideos(c);
}

@Override
public void onLoaderReset(Loader<Cursor> loader) {
  ((VideoAdapter)getAdapter()).setVideos(null);
}
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

VideoAdapter is another subclass of RecyclerView.Adapter, this time with smarts for dealing with a Cursor as the source of model data:

```
class VideoAdapter extends RecyclerView.Adapter<RowController> {
  Cursor videos=null;

  @Override
  public RowController onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowController(getLayoutInflater()
                             .inflate(R.layout.row, parent, false)));
  }

  void setVideos(Cursor videos) {
    this.videos=videos;
    notifyDataSetChanged();
  }

  @Override
  public void onBindViewHolder(RowController holder, int position) {
    videos.moveToPosition(position);
    holder.bindModel(videos);
  }
```

```
    @Override
    public int getItemCount() {
      if (videos==null) {
        return(0);
      }

      return(videos.getCount());
    }
  }
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

Specifically:

- `getItemCount()` returns the count of videos from the `Cursor`, or `0` if the `Cursor` is `null` (mimicking the behavior of `CursorAdapter`, which also treats a `null` `Cursor` as merely being one that has no rows)
- `onCreateViewHolder()` creates the `RowController`
- `onBindViewHolder()` moves the `Cursor` to the desired position, then passes the `Cursor` over to the `RowController`

Also note that we have a `setVideos()` method that is used to associate our `Cursor` of video information with the adapter. This also triggers a call to `notifyDataSetChanged()`, to ensure that the `RecyclerView` knows that our model has changed and it should re-render its contents.

The `RowController` constructor retrieves the necessary widgets from the row and setting up an `OnClickListener`:

```
  RowController(View row) {
    super(row);

    title=(TextView)row.findViewById(android.R.id.text1);
    thumbnail=(ImageView)row.findViewById(R.id.thumbnail);

    row.setOnClickListener(this);
  }
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/RowController.java)

The `bindModel()` method invoked by `onBindViewHolder()` on `VideoAdapter` uses the same basic logic from the original `VideoList` sample to populate the row widgets, plus holds onto the `Uri` and MIME type of the video in data members for the current row:

```
void bindModel(Cursor row) {
  title.setText(row.getString(
    row.getColumnIndex(MediaStore.Video.Media.TITLE)));

  videoUri=
      ContentUris.withAppendedId(
        MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
        row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID)));

  Picasso.with(thumbnail.getContext())
    .load(videoUri.toString())
    .fit().centerCrop()
    .placeholder(R.drawable.ic_media_video_poster)
    .into(thumbnail);

  int uriColumn=row.getColumnIndex(MediaStore.Video.Media.DATA);
  int mimeTypeColumn=
      row.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);

  videoMimeType=row.getString(mimeTypeColumn);
}
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/RowController.java)

The `onClick()` method uses those saved `Uri` and MIME type values for starting up the activity to play the selected video:

```
@Override
public void onClick(View v) {
  Intent i=new Intent(Intent.ACTION_VIEW);

  i.setDataAndType(videoUri, videoMimeType);
  title.getContext().startActivity(i);
}
```

(from RecyclerView/VideoList/app/src/main/java/com/commonsware/android/recyclerview/videolist/RowController.java)

Other than the lack of dividers, the UI is very similar to the original `VideoList`.

This sample app is used as the basis for many other samples in this book, such as the drag-and-drop examples.

Grids

So far, we have focused on one visual representation of our collection of model data: a vertically-scrolling list. In the `AdapterView` family, a given `AdapterView` subclass

has a specific visual representation (`ListView` for a vertically-scrolling list, `GridView` for a two-dimensional grid, etc.). With `RecyclerView`, the choice of layout manager determines most of the visual representation, and so switching from a list to a grid can be as simple as a single-line change to our code.

The key, though, is the word *can* in the previous sentence. Depending upon what you want to do, a grid-styled `RecyclerView` can be more complicated, simply because you now have two dimensions' worth of power and configuration to play with.

A Simple Grid

Making a `RecyclerView` use a grid is a matter of swapping out `LinearLayoutManager` for `GridLayoutManager`. In the RecyclerView/Grid sample project, you will see a clone of the `CardRippleList3` sample app, where we are now using `GridLayoutManager` in `onCreate()` of `MainActivity`:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  setLayoutManager(new GridLayoutManager(this, 2));
  setAdapter(new IconicAdapter());
}
```

(from RecyclerView/Grid/app/src/main/java/com/commonsware/android/recyclerview/grid/MainActivity.java)

`GridLayoutManager` takes a number of "spans", as well as a `Context`, as constructor parameters. In the simple case, as is with this app, "spans" will equate to "columns": each item returned by the `RecyclerView.Adapter` will go into a single-row, single-span cell.

In our case, we requested two spans, and so our result resides in two columns:

Figure 540: Grid RecyclerView Demo

In this case, this is a "true" grid, with rows and columns of cells. Hence, the height of a row is determined by the tallest cell in that row. The "amet" cell in the left column of the third row is taller than required because of the word-wrap of the "consectetuer" cell in the right column of the same row, for example. Later in this chapter, we will examine yet another option, `StaggeredGridLayoutManager`, where cells do not necessarily line up neatly in rows.

Choosing the Number of Columns

If we rotate the screen for the above sample, you will see that the cells fit a bit better, since they are really repurposed list-style rows:

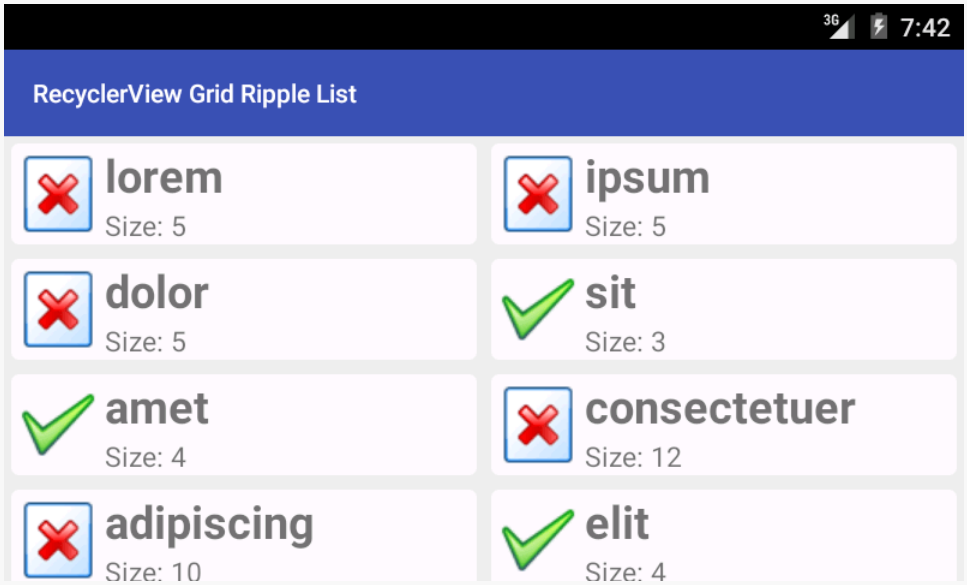


Figure 541: Grid RecyclerView Demo, Landscape

However, some apps may have smaller per-cell content. Plus, we have tablets to consider, and perhaps even televisions. It may be that you want to determine how many spans to use based on screen size and orientation.

One approach for doing that would be to use integer resources. You could have a `res/values/ints.xml` file with `<integer>` elements, giving the integer a name (`name` attribute) and value (text of the `<integer>` node). You could also have `res/values-w600dp/ints.xml` or other variations of the resource, where you provide different values to use for different screen sizes. Then, at runtime, call `getResources().getInteger()` to retrieve the correct value of the resource to use for the current device, and use that in your `GridLayoutManager` constructor. Now, you are in control over how many columns there are, by controlling how many spans are supplied to the constructor.

Another approach, suggested by Chiu-Ki Chan, is to create a subclass of `RecyclerView`, on which you provide a custom attribute for a desired approximate column width. Then, in your subclass' `onMeasure()` method, you can *calculate* the number of spans to use to give you the desired column width.

Of course, another way to take advantage of screen space is to grow the cells. By default, they will grow evenly, as each cell takes up one span, and the spans are evenly sizes. However, you can change that behavior, by attaching a `GridLayoutManager.SpanSizeLookup` to the `GridLayoutManager`. The

`GridLayoutManager.SpanSizeLookup` is responsible for indicating, for a given item's `position`, how many spans it should take up in the grid. We will examine how this works later in this chapter.

Varying the Items

So far, all of the items in the `RecyclerView` have had the same basic structure, just with varying content in the widgets in those items. But, it is entirely possible that we will want to have some items be more substantively different, based on different layouts. `ListView` and kin handle this via `getViewTypeCount()` and `getItemViewType()` in the `ListAdapter`. `RecyclerView` and `RecyclerView.Adapter` offer a similar mechanism, including their own variant of the `getItemViewType()` method. In this section, we will examine how this works, both with lists and grids.

A List with Headers

There are many cases where we want to have a list with some sort of section headers. The look of the headers usually is substantially different than the look of the rest of the rows, and therefore the best way to handle this is to teach the adapter about multiple row types.

This can be seen in the `RecyclerView/HeaderList` sample project. This is a clone of a similar project for `ListView`, where we want to put the 25 Latin words into 5 groups of 5 words each, with each group getting its own header.

Hence, our model data is now a two-dimensional `String` array:

```
private static final String[][] items= {
    { "lorem", "ipsum", "dolor", "sit", "amet" },
    { "consectetuer", "adipiscing", "elit", "morbi", "vel" },
    { "ligula", "vitae", "arcu", "aliquet", "mollis" },
    { "etiam", "vel", "erat", "placerat", "ante" },
    { "porttitor", "sodales", "pellentesque", "augue", "purus" } };
```

(from RecyclerView/HeaderList/app/src/main/java/com/commonsware/android/recyclerview/headerlist/MainActivity.java)

Our `getItemCount()` method now needs to take into account the headers, as well as the regular rows. There is one header row per batch of items, and so `getItemCount()` sums up the sizes of the batches with the extra header rows:

```
  @Override
  public int getItemCount() {
```

```
    int count=0;

    for (String[] batch : items) {
      count+=1 + batch.length;
    }

    return(count);
  }
```

(from RecyclerView/HeaderList/app/src/main/java/com/commonsware/android/recyclerview/headerlist/MainActivity.java)

In order to teach `RecyclerView` about our different rows, we need to implement `getItemViewType()`. Unlike its counterpart on `ListAdapter`, `getItemViewType()` can return *any* `int` value, so long as it is unique for the row type. In fact, the recommendation is to use dedicated ID resources to ensure that uniqueness.

To that end, we define two ID resources, in a `res/values/ids.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <item type="id" name="header"/>
  <item type="id" name="detail"/>
</resources>
```

(from RecyclerView/HeaderList/app/src/main/res/values/ids.xml)

Then, `getItemViewType()` can return `R.id.header` or `R.id.detail` to identify the two row types, and specifically which row type corresponds to the supplied `position`:

```
  @Override
  public int getItemViewType(int position) {
    if (getItem(position) instanceof Integer) {
      return(R.id.header);
    }

    return(R.id.detail);
  }

  private Object getItem(int position) {
    int offset=position;
    int batchIndex=0;

    for (String[] batch : items) {
      if (offset == 0) {
        return(Integer.valueOf(batchIndex));
      }
```

```
      offset--;

      if (offset < batch.length) {
        return(batch[offset]);
      }

      offset-=batch.length;
      batchIndex++;
    }

    throw new IllegalArgumentException("Invalid position: "
        + String.valueOf(position));
  }
```

(from RecyclerView/HeaderList/app/src/main/java/com/commonsware/android/recyclerview/headerlist/MainActivity.java)

This leverages a copy of the `getItem()` method from the original `ListView` version of this sample, which returns an `Integer` for a header item (identifying which header it is) and a `String` for detail item (identifying what Latin word to use). Note that `getItem()` is not part of the `RecyclerView.Adapter` protocol, but you are certainly welcome to have one if you want it.

In `onCreateViewHolder()`, we can now start paying attention to the second parameter, which we have been studiously ignoring until now. That value, `viewType`, will be a value that we returned from `getItemViewType()`, and it indicates what sort of `RecyclerView.ViewHolder` we should return. In our case, there are only two possibilities, and so we just inflate the appropriate layout and use a dedicated controller class (`HeaderController` for headers, `RowController` for detail):

```
  @Override
  public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    if (viewType==R.id.detail) {
      return(new RowController(getLayoutInflater()
                  .inflate(R.layout.row, parent, false)));
    }

    return(new HeaderController(getLayoutInflater()
                .inflate(R.layout.header, parent, false)));
  }
```

(from RecyclerView/HeaderList/app/src/main/java/com/commonsware/android/recyclerview/headerlist/MainActivity.java)

Similarly, our binding logic in `onBindViewHolder()` needs to route the right sort of model information to the proper controller:

```
  @Override
  public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
```

```
    if (holder instanceof RowController) {
      ((RowController)holder).bindModel((String)getItem(position));
    }
    else {
      ((HeaderController)holder).bindModel((Integer)getItem(position));
    }
  }
```

(from RecyclerView/HeaderList/app/src/main/java/com/commonsware/android/recyclerview/headerlist/MainActivity.java)

`RowController` is the same sort of setup as we have had in past examples. `HeaderController` is too, though it is far simpler, as we have only one widget needing to be updated (a `TextView` named `label`) and we do not care about click events:

```
package com.commonsware.android.recyclerview.headerlist;

import android.support.v7.widget.RecyclerView;
import android.view.View;
import android.widget.TextView;

class HeaderController extends RecyclerView.ViewHolder {
  TextView label=null;
  String template=null;

  HeaderController(View row) {
    super(row);

    label=(TextView)row.findViewById(R.id.label);

    template=label.getContext().getString(R.string.header_template);
  }

  void bindModel(Integer headerIndex) {
    label.setText(String.format(template, headerIndex.intValue()+1));
  }
}
```

(from RecyclerView/HeaderList/app/src/main/java/com/commonsware/android/recyclerview/headerlist/HeaderController.java)

The results are header rows with one look-and-feel, and detail rows with a different look-and-feel:

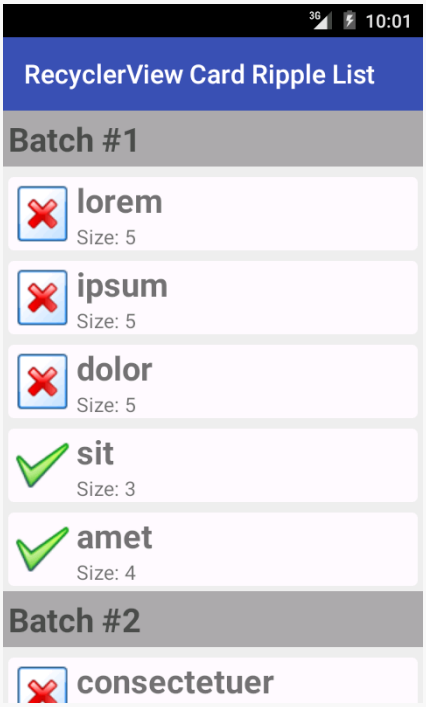


Figure 542: HeaderList RecyclerView Demo

A Grid-Style Table

In the discussion of `RecyclerView` grids, we saw that one way to take advantage of larger screens is to have more cells, in part by having more spans across the screen.

Another way to take advantage of screen space is to grow the cells. By default, they will grow evenly, as each cell takes up one span, and the spans are evenly sizes. However, you can change that behavior, by attaching a `GridLayoutManager.SpanSizeLookup` to the `GridLayoutManager`. The `GridLayoutManager.SpanSizeLookup` is responsible for indicating, for a given item's `position`, how many spans it should take up in the grid.

One way of employing a `GridLayoutManager.SpanSizeLookup` is to make a table. If you want a table, but the user should only be able to select rows, that would be a matter of using a `LinearLayoutManager` and setting up the rows with "cells" that are of consistent size per row. For example, each row could be a horizontal `LinearLayout`, where the "column" widths are determined using `android:layout_weight`. But sometimes you want a table where individual cells can be clicked upon (or selected via a five-way navigation option, like a trackball). In this case, `GridLayoutManager.SpanSizeLookup` will let you indicate, for a "column" of

your output, how many spans the cell should take up. By using a consistent number of spans for each column, you can get the same sort of weighted column width that you might get with `LinearLayout`-based rows in a `LinearLayoutManager`-powered `RecyclerView`.

And that will make a lot more sense (hopefully) when you see an example.

The `RecyclerView/VideoTable` sample project is a clone of the `VideoList` sample project from earlier in the chapter, with a few changes:

- We are going to use a `GridLayoutManager`, yet still organize our output into logical rows, by having three cells per row (title, thumbnail, and video duration)
- We are going to use `GridLayoutManager.SpanSizeLookup` to control the widths of each column in our grid
- Because our cells have varying content (`ImageView` in one, `TextView` in others), we will use different controllers for those cells, each optimized for handling that cell's sort of content

The two columns that will hold text (title and video duration) will use the following layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:padding="8dp"
  android:background="?android:attr/selectableItemBackground">

  <TextView
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_gravity="center_vertical"
    android:textSize="24sp"/>
</LinearLayout>
```

(from RecyclerView/VideoTable/app/src/main/res/layout/label.xml)

The `LinearLayout` root element may seem superfluous, but we are using it for the `selectableItemBackground`, to provide a response when the cell is clicked upon.

Similarly, we have a layout dedicated to the thumbnail:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:padding="8dp"
  android:background="?android:attr/selectableItemBackground">

  <ImageView
    android:id="@+id/thumbnail"
    android:layout_width="96dp"
    android:layout_height="72dp"
    android:contentDescription="@string/thumbnail"/>

</LinearLayout>
```

(from RecyclerView/VideoTable/app/src/main/res/layout/thumbnail.xml)

`onCreate()` of `MainActivity` is largely the same as before. This time, though, we are creating an instance of a `ColumnWeightSpanSizeLookup` class and using it for two things:

1. Calling its `getTotalSpans()` to tell the `GridLayoutManager` how many spans to use
2. Using it as a `GridLayoutManager.SpanSizeLookup`, attaching it to the `GridLayoutManager` via `setSpanSizeLookup()`:

```
    ColumnWeightSpanSizeLookup spanSizer=new ColumnWeightSpanSizeLookup(COLUMN_WEIGHTS);
    GridLayoutManager mgr=new GridLayoutManager(this, spanSizer.getTotalSpans());

    mgr.setSpanSizeLookup(spanSizer);
    setLayoutManager(mgr);
    setAdapter(new VideoAdapter());
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/MainActivity.java)

The latter point means that `ColumnWeightSpanSizeLookup` is a subclass of the abstract `GridLayoutManager.SpanSizeLookup` base class. The one method that you need to override in a `GridLayoutManager.SpanSizeLookup` subclass is `getSpanSize()`. Given an item's `position`, `getSpanSize()` returns the number of spans that the item's cell should... um... span.

(we overload the word "span" a lot in Android...)

`ColumnWeightSpanSizeLookup` handles this via a set of column weights, which it gets as an `int` array in the constructor. `onCreate()` referenced a `COLUMN_WEIGHTS` `static` data member for the weights:

```
  private static final int[] COLUMN_WEIGHTS={1, 4, 1};
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/MainActivity.java)

This `int` array tells us both how many columns there are and how wide each column should be, in terms of spans.

Converting the `position` to a column index is a matter of applying the modulo (`%`) operator, so the implementation of `getSpanSize()` on `ColumnWeightSpanSizeLookup` just returns the `columnWeights` value for the desired column:

```
package com.commonsware.android.recyclerview.videotable;

import android.support.v7.widget.GridLayoutManager;

class ColumnWeightSpanSizeLookup extends GridLayoutManager.SpanSizeLookup {
  private final int[] columnWeights;

  ColumnWeightSpanSizeLookup(int[] columnWeights) {
    this.columnWeights=columnWeights;
  }

  @Override
  public int getSpanSize(int position) {
    return(columnWeights[position % columnWeights.length]);
  }

  int getTotalSpans() {
    int sum=0;

    for (int weight : columnWeights) {
      sum+=weight;
    }

    return(sum);
  }
}
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/ColumnWeightSpanSizeLookup.java)

`getTotalSpans()` is a convenience method, to sum all of the column weights. That is how many spans the `GridLayoutManager` will use overall, with each column getting its specific number of spans based upon the `int` array. Note that while we hard-coded the `int` array values in this case, there is nothing stopping us from using

<integer-array> resources to pull these values out of the Java code, and perhaps even vary them by screen size or other configuration variations.

All of that will set up our grid with the correct number of spans and the right number of spans to use per column of the output. The combination will give us the row structure, as each row's worth of columns uses all of the spans for that row, forcing GridLayoutManager to put subsequent items on the next row.

The rest of the project is focused on having different widgets for those different cells, using getItemViewType() and so on.

The VideoAdapter implementation of getItemViewType() simply returns the position modulo 3, to return a unique value (in this case, 0, 1, or 2):

```
  @Override
  public int getItemViewType(int position) {
    return(position % 3);
  }
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/MainActivity.java)

getItemCount() takes into account that there are three cells per video, and so the number of items being managed by this adapter is triple the number of videos:

```
  @Override
  public int getItemCount() {
    if (videos==null) {
      return(0);
    }

    return(videos.getCount()*3);
  }
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/MainActivity.java)

The onCreateViewHolder() and onBindViewHolder() methods take into account those three item types, using a VideoThumbnailController or a VideoTextController depending on the item type. Both of those classes will inherit from a BaseVideoController, which defines a bindModel() method that onBindViewHolder() can use:

```
  @Override
  public BaseVideoController onCreateViewHolder(ViewGroup parent, int viewType) {
    BaseVideoController result=null;

    switch(viewType) {
```

```
      case 0:
        result=new VideoThumbnailController(getLayoutInflater()
                                               .inflate(R.layout.thumbnail,
                                                        parent, false));
        break;

      case 1:
        int cursorColumn=videos.getColumnIndex(MediaStore.Video.VideoColumns.DISPLAY_NAME);

        result=new VideoTextController(getLayoutInflater()
                                         .inflate(R.layout.label,
                                             parent, false),
                                       android.R.id.text1,
                                       cursorColumn);
        break;

      case 2:
        cursorColumn=videos.getColumnIndex(MediaStore.Video.VideoColumns.DURATION);

        result=new VideoTextController(getLayoutInflater()
                                         .inflate(R.layout.label,
                                             parent, false),
                                       android.R.id.text1,
                                       cursorColumn);
        break;
    }

    return(result);
  }

  @Override
  public void onBindViewHolder(BaseVideoController holder, int position) {
    videos.moveToPosition(position/3);
    holder.bindModel(videos);
  }
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/MainActivity.java)

`BaseVideoController` handles click events on the cell, along with collecting the `Uri` and MIME type of the video to use on click events:

```
package com.commonsware.android.recyclerview.videotable;

import android.content.ContentUris;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.provider.MediaStore;
import android.support.v7.widget.RecyclerView;
import android.view.View;

abstract class BaseVideoController extends RecyclerView.ViewHolder
    implements View.OnClickListener {
  private Uri videoUri=null;
  private String videoMimeType=null;
```

```
  BaseVideoController(View cell) {
    super(cell);

    cell.setOnClickListener(this);
  }

  @Override
  public void onClick(View v) {
    Intent i=new Intent(Intent.ACTION_VIEW);

    i.setDataAndType(videoUri, videoMimeType);
    itemView.getContext().startActivity(i);
  }

  void bindModel(Cursor row) {
    int mimeTypeColumn=
        row.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);

    videoUri=ContentUris.withAppendedId(
      MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
      row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID)));
    videoMimeType=row.getString(mimeTypeColumn);
  }
}
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/BaseVideoController.java)

`VideoTextController` extends `BaseVideoController` and handles binding some column from the `MediaStore` `Cursor` to a `TextView` with some ID:

```
package com.commonsware.android.recyclerview.videotable;

import android.database.Cursor;
import android.view.View;
import android.widget.TextView;

class VideoTextController extends BaseVideoController {
  private TextView label=null;
  private int cursorColumn;

  VideoTextController(View cell, int labelId, int cursorColumn) {
    super(cell);
    this.cursorColumn=cursorColumn;

    label=(TextView)cell.findViewById(labelId);
  }
```

```
  @Override
  void bindModel(Cursor row) {
    super.bindModel(row);

    label.setText(row.getString(cursorColumn));
  }
}
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/VideoTextController.java)

`VideoThumbnailController` handles using Picasso to get the video thumbnail asynchronously and binding it to an `ImageView` in the inflated cell `View`:

```
package com.commonsware.android.recyclerview.videotable;

import android.content.ContentUris;
import android.database.Cursor;
import android.net.Uri;
import android.provider.MediaStore;
import android.view.View;
import android.widget.ImageView;
import com.squareup.picasso.Picasso;

class VideoThumbnailController extends BaseVideoController {
  private ImageView thumbnail=null;

  VideoThumbnailController(View cell) {
    super(cell);

    thumbnail=(ImageView)cell.findViewById(R.id.thumbnail);
  }

  @Override
  void bindModel(Cursor row) {
    super.bindModel(row);

    Uri video=
        ContentUris.withAppendedId(
          MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
          row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID)));

    Picasso.with(thumbnail.getContext())
      .load(video.toString())
      .fit().centerCrop()
      .placeholder(R.drawable.ic_media_video_poster)
      .into(thumbnail);
  }
}
```

(from RecyclerView/VideoTable/app/src/main/java/com/commonsware/android/recyclerview/videotable/VideoThumbnailController.java)

The result is the same information as was in the original `VideoList` demo, but organized into a table, where each cell is clickable:

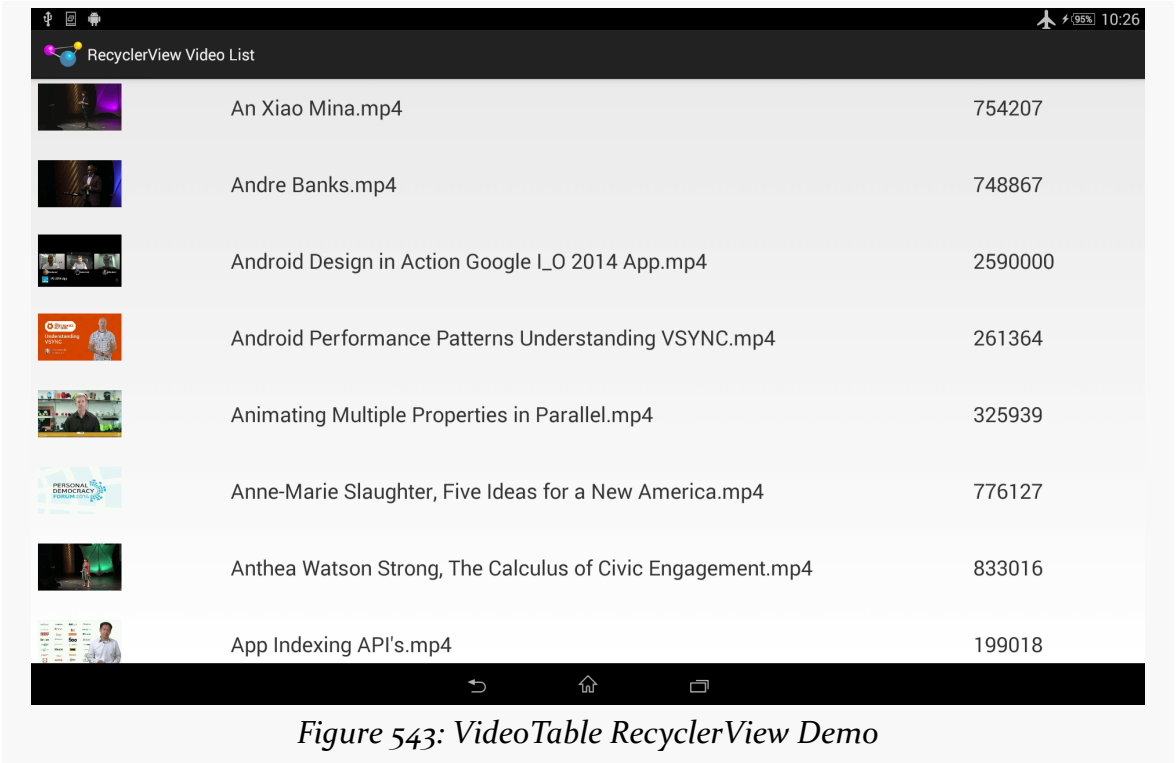


Figure 543: VideoTable RecyclerView Demo

The duration is returned by `MediaStore` in milliseconds, which is not a great choice to present directly to the user. An improved version of this app might use a dedicated `RecyclerView.ViewHolder` that would convert the millisecond count into a duration measured in hours, minutes, and seconds (e.g., shown as `HH:MM:SS` to the user).

Also note that the cell sizes are purely driven by their weights, which will not necessarily handle all content in all configurations very well. The chosen weights barely work on a 10" tablet in portrait, for example:

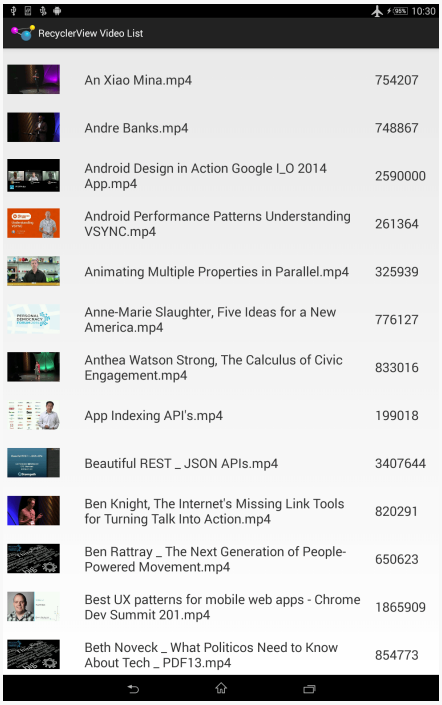


Figure 544: VideoTable RecyclerView Demo, Portrait

Mutable Row Contents

So far, all of the items we have used have been display-only. At most, they might respond to click events, along the lines of clicking a `ListView` row or `GridView` cell.

But, what about choice modes?

`ListView` and `GridView` — by way of their common `AbsListView` ancestor – have the concept of choice modes, where the user can "check" and "uncheck" items, and the list or grid will keep track of those states.

Well, as with lots of other things involving `RecyclerView`, `RecyclerView` does not offer choice modes... though you can implement that yourself. The `RecyclerView/ChoiceList` sample project turns our list-style `RecyclerView` into a checklist, with `CheckBox` widgets in each row, where the `RecyclerView.Adapter` will keep track of the `CheckBox` checked states for us.

First, we need to add a `CheckBox` to the row:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:cardview="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="4dp"
  cardview:cardCornerRadius="4dp">

  <LinearLayout
    android:id="@+id/row_content"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:background="?android:attr/selectableItemBackground">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:padding="2dip"
      android:src="@drawable/ok"
      android:contentDescription="@string/icon"/>

    <LinearLayout
      android:layout_width="0dip"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:orientation="vertical">

      <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="25sp"
        android:textStyle="bold"/>

      <TextView
        android:id="@+id/size"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="15sp"/>
    </LinearLayout>

    <CheckBox
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
```

```
      android:id="@+id/cb"
      android:layout_gravity="center_vertical"/>

  </LinearLayout>
</android.support.v7.widget.CardView>
```

(from RecyclerView/ChoiceList/app/src/main/res/layout/row.xml)

Our `IconicAdapter` is only slightly different than before:

- it inherits from a `ChoiceCapableAdapter` that we will examine shortly, and
- it supplies a `MultiChoiceMode` instance to `ChoiceCapableAdapter` as part of chaining to the `ChoiceCapableAdapter` constructor

```
  class IconicAdapter extends ChoiceCapableAdapter<RowController> {
    IconicAdapter() {
      super(new MultiChoiceMode());
    }

    @Override
    public RowController onCreateViewHolder(ViewGroup parent, int viewType) {
      return(new RowController(this, getLayoutInflater()
                                       .inflate(R.layout.row, parent, false)));
    }

    @Override
    public void onBindViewHolder(RowController holder, int position) {
      holder.bindModel(items[position]);
    }

    @Override
    public int getItemCount() {
      return(items.length);
    }
  }
```

(from RecyclerView/ChoiceList/app/src/main/java/com/commonsware/android/recyclerview/choicelist/MainActivity.java)

`ChoiceCapableAdapter` is simply a `RecyclerView.Adapter` that knows how to handle choice modes, as implemented via the `ChoiceMode` interface:

```
package com.commonsware.android.recyclerview.choicelist;

import android.os.Bundle;

public interface ChoiceMode {
  void setChecked(int position, boolean isChecked);
```

```
  boolean isChecked(int position);
  void onSaveInstanceState(Bundle state);
  void onRestoreInstanceState(Bundle state);
}
```

(from RecyclerView/ChoiceList/app/src/main/java/com/commonsware/android/recyclerview/choicelist/ChoiceMode.java)

A `ChoiceMode` is effectively a strategy class, responsible for tracking the checked states, not only for the current `ChoiceCapableAdapter` instance, but for future ones created as part of a configuration change. It requires four methods:

- `setChecked()` and `isChecked()` are getters and setters for whether or not a given `position` is checked
- `onSaveInstanceState()` and `onRestoreInstanceState()` manage storing and restoring those check states from the saved instance state `Bundle` of an activity or fragment

This project uses a `MultiChoiceMode` implementation of `ChoiceMode`:

```
package com.commonsware.android.recyclerview.choicelist;

import android.os.Bundle;

public class MultiChoiceMode implements ChoiceMode {
  private static final String STATE_CHECK_STATES="checkStates";
  private ParcelableSparseBooleanArray checkStates=new ParcelableSparseBooleanArray();

  @Override
  public void setChecked(int position, boolean isChecked) {
    checkStates.put(position, isChecked);
  }

  @Override
  public boolean isChecked(int position) {
    return(checkStates.get(position, false));
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    state.putParcelable(STATE_CHECK_STATES, checkStates);
  }

  @Override
  public void onRestoreInstanceState(Bundle state) {
    checkStates=state.getParcelable(STATE_CHECK_STATES);
  }
}
```

(from RecyclerView/ChoiceList/app/src/main/java/com/commonsware/android/recyclerview/choicelist/MultiChoiceMode.java)

`MultiChoiceMode`, in turn, is mostly handled by a `ParcelableSparseBooleanArray`. `SparseBooleanArray` is a class, supplied in the Android SDK, that is a space-efficient

mapping of `int` values to `boolean` values, as opposed to using a `HashMap` and having to convert those primitives to `Integer` and `Boolean` objects. However, for inexplicable reasons, `SparseBooleanArray` was not implemented to be `Parcelable`, and therefore it cannot be stored in a `Bundle`. `ParcelableSparseBooleanArray` is a subclass of `SparseBooleanArray` that handles the `Parcelable` aspects:

```
package com.commonsware.android.recyclerview.choicelist;

import android.os.Parcel;
import android.os.Parcelable;
import android.util.SparseBooleanArray;

public class ParcelableSparseBooleanArray extends SparseBooleanArray
    implements Parcelable {
  public static Parcelable.Creator<ParcelableSparseBooleanArray> CREATOR
    =new Parcelable.Creator<ParcelableSparseBooleanArray>() {
    @Override
    public ParcelableSparseBooleanArray createFromParcel(Parcel source) {
      return(new ParcelableSparseBooleanArray(source));
    }

    @Override
    public ParcelableSparseBooleanArray[] newArray(int size) {
      return(new ParcelableSparseBooleanArray[size]);
    }
  };

  public ParcelableSparseBooleanArray() {
    super();
  }

  private ParcelableSparseBooleanArray(Parcel source) {
    int size=source.readInt();

    for (int i=0; i < size; i++) {
      put(source.readInt(), (Boolean)source.readValue(null));
    }
  }

  @Override
  public int describeContents() {
    return(0);
  }

  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeInt(size());
```

```
    for (int i=0;i<size();i++) {
      dest.writeInt(keyAt(i));
      dest.writeValue(valueAt(i));
    }
  }
}
```

(from RecyclerView/ChoiceList/app/src/main/java/com/commonsware/android/recyclerview/choicelist/ParcelableSparseBooleanArray.java)

The net effect is that `MultiChoiceMode`, by means of `ParcelableSparseBooleanArray`, can track the checked/unchecked states of particular item `position` values.

`ChoiceCapableAdapter`, then, is a `RecyclerView.ViewHolder` that surfaces a `ChoiceMode` implementation:

```
package com.commonsware.android.recyclerview.choicelist;

import android.os.Bundle;
import android.support.v7.widget.RecyclerView;

abstract public class
    ChoiceCapableAdapter<T extends RecyclerView.ViewHolder>
    extends RecyclerView.Adapter<T> {
  private final ChoiceMode choiceMode;

  public ChoiceCapableAdapter(ChoiceMode choiceMode) {
    super();
    this.choiceMode=choiceMode;
  }

  void onChecked(int position, boolean isChecked) {
    choiceMode.setChecked(position, isChecked);
  }

  boolean isChecked(int position) {
    return(choiceMode.isChecked(position));
  }

  void onSaveInstanceState(Bundle state) {
    choiceMode.onSaveInstanceState(state);
  }

  void onRestoreInstanceState(Bundle state) {
    choiceMode.onRestoreInstanceState(state);
```

```
  }
}
```

(from RecyclerView/ChoiceList/app/src/main/java/com/commonsware/android/recyclerview/choicelist/ChoiceCapableAdapter.java)

The methods exposed by `ChoiceCapableAdapter` can then be used by outside parties. Specifically, `MainActivity` delegates `onSaveInstanceState()` and `onRestoreInstanceState()` to `ChoiceCapableAdapter`, so checked states can span configuration changes and the like. Plus, `RowController` can hook up on `OnCheckedChangedListener` and to update `ChoiceCapableAdapter` based on the state of checkbox changes:

```
package com.commonsware.android.recyclerview.choicelist;

import android.annotation.TargetApi;
import android.os.Build;
import android.support.v7.widget.RecyclerView;
import android.view.MotionEvent;
import android.view.View;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;

class RowController extends RecyclerView.ViewHolder
    implements View.OnClickListener, CompoundButton.OnCheckedChangeListener {
  private ChoiceCapableAdapter adapter;
  private TextView label=null;
  private TextView size=null;
  private ImageView icon=null;
  private String template=null;
  private CheckBox cb=null;

  RowController(ChoiceCapableAdapter adapter, View row) {
    super(row);

    this.adapter=adapter;
    label=(TextView)row.findViewById(R.id.label);
    size=(TextView)row.findViewById(R.id.size);
    icon=(ImageView)row.findViewById(R.id.icon);
    cb=(CheckBox)row.findViewById(R.id.cb);

    template=size.getContext().getString(R.string.size_template);

    row.setOnClickListener(this);
```

```
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
      row.setOnTouchListener(new View.OnTouchListener() {
        @TargetApi(Build.VERSION_CODES.LOLLIPOP)
        @Override
        public boolean onTouch(View v, MotionEvent event) {
          v
              .findViewById(R.id.row_content)
              .getBackground()
              .setHotspot(event.getX(), event.getY());

          return(false);
        }
      });
    }

    cb.setOnCheckedChangeListener(this);
  }

  @Override
  public void onClick(View v) {
    Toast.makeText(v.getContext(),
        String.format("Clicked on position %d", getAdapterPosition()),
        Toast.LENGTH_SHORT).show();
  }

  @Override
  public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
    adapter.onChecked(getAdapterPosition(), isChecked);
  }

  void bindModel(String item) {
    label.setText(item);
    size.setText(String.format(template, item.length()));

    if (item.length()>4) {
      icon.setImageResource(R.drawable.delete);
    }
    else {
      icon.setImageResource(R.drawable.ok);
    }

    cb.setChecked(adapter.isChecked(getAdapterPosition()));
  }
}
```

(from RecyclerView/ChoiceList/app/src/main/java/com/commonsware/android/recyclerview/choicelist/RowController.java)

Here, `bindModel()` updates the `CheckBox` based upon the `ChoiceCapableAdapter` `isChecked()` value for the `RecyclerView.ViewHolder` `position` (obtained via `getPosition()`). And, `onCheckedChanged()` updates the `ChoiceCapableAdapter` to keep track of whether this position is checked or unchecked, to handle row recycling, configuration changes, etc.

The result is much as you would expect: a version of our same sort of UI as before, except that if the user clicks the `CheckBox`, instead of the rest of the row, the `CheckBox` toggles its checked state, and that state survives row recycling, configuration changes, and so on:

Figure 545: ChoiceList RecyclerView Demo

Note that since this sample is using `Theme.Material` on Android 5.0+ devices, and since the screenshot is from an Android 5.0 emulator, the `CheckBox` styling is based on the accent color, here shown as bright yellow.

Switching to the Activated Style

Also note that `ChoiceCapableAdapter`, `MultiChoiceMode`, and kin are oblivious to *how* the user is informed about what is checked and unchecked. `RowController` in

the previous sample happens to use a `CheckBox`. `RowController` could use some other widget, like a `Switch`.

Another approach is to use the activated state. Once again, this is the sort of thing that is automatically handled for us by `ListView` and its choice modes, but with some minor tweaks, we can get our `RowController` to use this approach. This is shown in the `RecyclerView/ActivatedList` sample project.

First, we need to give our row a background that has a `StateListDrawable` that supports the activated state. The simplest approach — and the one traditionally used with `ListView` — is to set up an `activated` style with the stock theme-supplied background drawable, then apply that style to the row.

So, this sample app defines `activated` in `res/values/styles.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <style name="Theme.Apptheme" parent="@android:style/Theme.Holo.Light.DarkActionBar">
  </style>

  <style name="activated" parent="Theme.Apptheme">
    <item name="android:background">?android:attr/activatedBackgroundIndicator</item>
  </style>

</resources>
```

(from RecyclerView/ActivatedList/app/src/main/res/values/styles.xml)

Note that `activated` inherits from `Theme.Apptheme`. This means that we will get the `Theme.Holo`-flavored background normally, but on API Level 21+, we will get the `Theme.Material`-flavored background, courtesy of a `res/values-v21/styles.xml` override of `Theme.Apptheme`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Theme.Apptheme" parent="android:Theme.Material.Light.DarkActionBar">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

(from RecyclerView/ActivatedList/app/src/main/res/values-v21/styles.xml)

Our row layout now dumps the `CardView` (whose own background may conflict with the activated one) and applies the `activated` style to the root `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  style="@style/activated">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:padding="2dip"
    android:src="@drawable/ok"
    android:contentDescription="@string/icon"/>

  <LinearLayout
    android:layout_width="0dip"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:orientation="vertical">

    <TextView
      android:id="@+id/label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/size"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

(from RecyclerView/ActivatedList/app/src/main/res/layout/row.xml)

The row also no longer has the `CheckBox`, as it is no longer needed.

`RowController` now uses the `OnClickListener` interface to respond to clicks and use that to toggle the activated state for that row:

```
@Override
public void onClick(View v) {
  boolean isCheckedNow=adapter.isChecked(getAdapterPosition());

  adapter.onChecked(getAdapterPosition(), !isCheckedNow);
  row.setActivated(!isCheckedNow);
}
```

(from RecyclerView/ActivatedList/app/src/main/java/com/commonsware/android/recyclerview/activatedlist/RowController.java)

`setActivated()`, applied to a `View`, indicates that it is (or is not) activated, affecting anything in that `View` that depends upon that state, such as the background.

Similarly, `bindModel()` uses `setActivated()` to update the activated state when binding our data:

```
void bindModel(String item) {
  label.setText(item);
  size.setText(String.format(template, item.length()));

  if (item.length()>4) {
    icon.setImageResource(R.drawable.delete);
  }
  else {
    icon.setImageResource(R.drawable.ok);
  }

  row.setActivated(adapter.isChecked(getAdapterPosition()));
```

(from RecyclerView/ActivatedList/app/src/main/java/com/commonsware/android/recyclerview/activatedlist/RowController.java)

Everything else is the same as the original `CheckBox` version of the sample. But now, the “checked” state is indicated by the activated highlight:

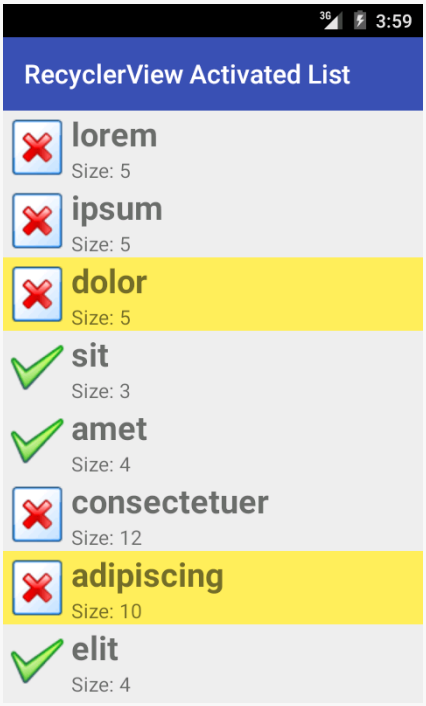


Figure 546: ActivatedList RecyclerView Demo

And, since this demo is running on Android 5.0, the activated highlight color is the accent color, which in this case is set to be yellow.

But, What About Single-Choice?

Both of the preceding examples illustrate multiple-choice behavior. Sometimes, though, single-choice behavior is the better option. For example, in a master-detail structure, in dual-pane mode (e.g., tablets, where the master and the detail are both visible), you probably normally want single-choice mode.

That is certainly possible, though, once again, `RecyclerView` does not offer it. It also adds a wrinkle: how do we arrange to uncheck a previously-checked item, when the user checks another item? Like `RadioButton` widgets in a `RadioGroup`, we need to ensure that only one item at a time is checked, and that will require us to update the UI of the formerly-checked-but-now-unchecked item.

With some tweaks, the last sample project, where we used the activated state for a multiple-choice list, can be revised to limit the user to a single choice. Those tweaks are illustrated in the `RecyclerView/SingleActivatedList` sample project.

The `ChoiceMode` interface now has two new methods:

1. `isSingleChoice()` will return `true` for a single-choice `ChoiceMode` strategy, `false` otherwise
2. `getCheckedPosition()` will return the position of whatever the currently-checked item is

```
package com.commonsware.android.recyclerview.singleactivatedlist;

import android.os.Bundle;

public interface ChoiceMode {
  boolean isSingleChoice();
  int getCheckedPosition();
  void setChecked(int position, boolean isChecked);
  boolean isChecked(int position);
  void onSaveInstanceState(Bundle state);
  void onRestoreInstanceState(Bundle state);
}
```

(from RecyclerView/SingleActivatedList/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/ChoiceMode.java)

`SingleChoiceMode` is now our implementation of `ChoiceMode`:

```
package com.commonsware.android.recyclerview.singleactivatedlist;

import android.os.Bundle;

public class SingleChoiceMode implements ChoiceMode {
  private static final String STATE_CHECKED="checkedPosition";
  private int checkedPosition=-1;

  @Override
  public boolean isSingleChoice() {
    return(true);
  }

  @Override
  public int getCheckedPosition() {
    return(checkedPosition);
  }

  @Override
  public void setChecked(int position, boolean isChecked) {
    if (isChecked) {
      checkedPosition=position;
```

```
    }
    else if (isChecked(position)) {
      checkedPosition=-1;
    }
  }

  @Override
  public boolean isChecked(int position) {
    return(checkedPosition==position);
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    state.putInt(STATE_CHECKED, checkedPosition);
  }

  @Override
  public void onRestoreInstanceState(Bundle state) {
    checkedPosition=state.getInt(STATE_CHECKED, -1);
  }
}
```

(from RecyclerView/SingleActivatedList/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/SingleChoiceMode.java)

`SingleChoiceMode` tracks the currently-checked position, using `-1` to indicate no position is checked. Of note, if a position was checked, then `setChecked()` unchecks it, `SingleChoiceMode` goes back to `-1` and indicates that there is no currently-checked position.

`ChoiceCapableAdapter` also has a couple of modifications. First, it now accepts the `RecyclerView` itself as a constructor parameter, holding onto it in an `rv` data member. And, `onChecked()` needs to be modified to take care of removing the activated state from whatever item had been previously checked when some new item is checked:

```
package com.commonsware.android.recyclerview.singleactivatedlist;

import android.os.Bundle;
import android.support.v7.widget.RecyclerView;

abstract public class
    ChoiceCapableAdapter<T extends RecyclerView.ViewHolder>
    extends RecyclerView.Adapter<T> {
  private final ChoiceMode choiceMode;
  private final RecyclerView rv;
```

```
  public ChoiceCapableAdapter(RecyclerView rv,
                              ChoiceMode choiceMode) {
    super();
    this.rv=rv;
    this.choiceMode=choiceMode;
  }

  void onChecked(int position, boolean isChecked) {
    if (choiceMode.isSingleChoice()) {
      int checked=choiceMode.getCheckedPosition();

      if (checked>=0) {
        RowController row=
            (RowController)rv.findViewHolderForAdapterPosition(checked);

        if (row!=null) {
          row.setChecked(false);
        }
      }
    }

    choiceMode.setChecked(position, isChecked);
  }

  boolean isChecked(int position) {
    return(choiceMode.isChecked(position));
  }

  void onSaveInstanceState(Bundle state) {
    choiceMode.onSaveInstanceState(state);
  }

  void onRestoreInstanceState(Bundle state) {
    choiceMode.onRestoreInstanceState(state);
  }

  @Override
  public void onViewAttachedToWindow(T holder) {
    super.onViewAttachedToWindow(holder);

    if (holder.getAdapterPosition()!=choiceMode.getCheckedPosition()) {
      ((RowController)holder).setChecked(false);
    }
  }
}
```

(from RecyclerView/SingleActivatedList/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/ChoiceCapableAdapter.java)

To do that, `onChecked()` asks the `ChoiceMode` if it is single choice. If yes, it gets the last checked position. If that position is plausible (0 or higher), it gets the `RecyclerView.ViewHolder` for that position via `findViewHolderForAdapterPosition()`, called on the `RecyclerView`. If this returns something other than `null`, then it must be a `RowController`, and so `onChecked()` calls `setChecked(false)` on that row to remove the activated state.

`findViewHolderForAdapterPosition()` and `findViewHolderForLayoutPosition()` replace the now-deprecated `findViewHolderForPosition()` method. All three methods do the same basic thing: given a position, return the `ViewHolder` for that position, if any. `findViewHolderForPosition()` and `findViewHolderForLayoutPosition()` have the same implementation, at least at the present time. The primary thing that `findViewHolderForAdapterPosition()` does differently is it always returns `null` if the data has been changed (e.g., `notifyDataSetChanged()` was called on the adapter) but those changes have not yet been laid out. In this sample app, that difference is academic, but `findViewHolderForAdapterPosition()` probably is a safer choice for most use cases.

However, these `find...()` methods have a wrinkle: they only return a `ViewHolder` if the row is *visible*. If the `ViewHolder` is cached, but not visible, `find...()` will still not return it. This causes a problem where we need to de-select a row that is not visible (and so `find...()` does not work) but will not be re-bound using `onBindViewHolder()` (as the `ViewHolder` is already set up). This requires us to implement `onViewAttachedToWindow()` — called whenever a `ViewHolder` contents are actually attached as children to the `RelativeLayout` — and update the checked state there, as a fallback.

(and many thanks to Mahmoud Abou-Eita for reporting that problem)

`setChecked()` did not exist in the previous sample, as the activated state was handled purely internally to `RowController`. So, now `RowController` has a `setChecked()` method to toggle the activated state:

```
void setChecked(boolean isChecked) {
  row.setActivated(isChecked);
}
```

(from RecyclerView/SingleActivatedList/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/RowController.java)

`MainActivity` now must supply the `RecyclerView` to the `IconicAdapter` in `onCreate()`:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  setLayoutManager(new LinearLayoutManager(this));
  adapter=new IconicAdapter(getRecyclerView());
  setAdapter(adapter);
}
```

(from RecyclerView/SingleActivatedList/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/MainActivity.java)

...so that `IconicAdapter()` can supply it to the `ChoiceCapableAdapter` superclass constructor:

```
  IconicAdapter(RecyclerView rv) {
    super(rv, new SingleChoiceMode());
  }
```

(from RecyclerView/SingleActivatedList/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/MainActivity.java)

Visually, the results are identical to the previous example, except that at most only one item can be checked at a time. The key is the phrase "at most" — this implementation allows the user to tap on a checked item to uncheck it. This may be fine, as your app may simply hide the detail in this scenario, still allowing the user to interact with action bar items (e.g., a "create new model" item). If you wanted to prevent that, have `SingleChoiceMode` *not* set `checkedPosition` to `-1` when the user taps on a previously-checked item, to leave the currently-checked position intact.

Keyboard Navigation

If you try using the `RecyclerView/SingleActivatedList` app — or any of the sample apps presented so far — on a device that has a physical keyboard or five-way navigation option (e.g., D-pad), you will find that `RecyclerView` has no built-in keyboard navigation. This is in contrast with standard `AdapterView` classes like `ListView`, where key events are handled automatically. Once again, if you want the behavior, you have to add it yourself.

The exact details of what you want to do when the user tries navigating with a keyboard will vary, based on lots of things:

- What sort of layout manager are you using? If it is a list, you only need to worry about moving up and down, while if it is a grid, you will need to handle movement along both axes.
- Do you have focusable widgets in your `RecyclerView` items? If so, how will navigation between those widgets blend with navigation through the `RecyclerView` overall?
- Are you using any sort of "selection" or "choice" model? If so, do you want keyboard navigation to change the choice? Or, do you want keyboard navigation to only make a choice when the user does something special (e.g., presses an `Enter` key or a center D-pad button), with the two-way or four-way navigation showing up as something separate from the user's choice?

The RecyclerView/SingleActivatedListKB sample project is a clone of the `SingleActivatedList` sample, except that we now support keyboard events. Specifically, the user can use the up and down arrow keys to change the selected row in the list. This is perhaps the simplest scenario:

- We have no focusable widgets, so key events can just change the selected row
- It is just a list, so we only need to worry about two directions, not four
- We want either zero or one selected row, and so we do not need to have a distinction between navigation and selection, as we might with a multiple-choice list

And, the best part is that we only need to change `ChoiceCapableAdapter` — the rest of the app can remain unchanged.

First, we override another method on `RecyclerView.Adapter`: `onAttachedToRecyclerView()`. As the name suggests, this method is called when our adapter is assigned to a `RecyclerView` instance. Here, if we are in single-choice mode, we register an `OnKeyListener` on the `RecyclerView` itself, to find out when it receives key events:

```
// inspired by http://stackoverflow.com/a/28838834/115145

@Override
public void onAttachedToRecyclerView(RecyclerView rv) {
  super.onAttachedToRecyclerView(rv);

  if (choiceMode.isSingleChoice()) {
    rv.setOnKeyListener(new View.OnKeyListener() {
      @Override
```

```
      public boolean onKey(View v, int keyCode, KeyEvent event) {
        if (event.getAction()==KeyEvent.ACTION_DOWN) {
          switch (keyCode) {
            case KeyEvent.KEYCODE_DPAD_DOWN:
              return(chooseNext());
            case KeyEvent.KEYCODE_DPAD_UP:
              return(choosePrevious());
          }
        }

        return(false);
      }
    });
  }
}
```

(from RecyclerView/SingleActivatedListKB/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/ChoiceCapableAdapter.java)

When the user presses down either the up or down arrow key (or equivalents on a D-pad), we call out to private `choosePrevious()` and `chooseNext()` methods, which will return `true` if we moved the selection in that direction, `false` otherwise. If the key event is not one of those, we return `false` to indicate that we are not consuming the key event.

The `choosePrevious()` and `chooseNext()` methods are responsible for determining what our next selection should be, assuming that the selection can change in the designated direction:

```
private boolean chooseNext() {
  long now=System.currentTimeMillis();
  boolean result=false;

  if (lastDownKeyTime==-1 || now-lastDownKeyTime>KEY_TIME_DELTA) {
    lastDownKeyTime=now;
    lastUpKeyTime=-1L;

    int checked=choiceMode.getCheckedPosition();

    if (checked<0) {
      onChecked(0, true, true);
      result=true;
    }
    else if (checked<getItemCount()-1) {
      onChecked(checked+1, true, true);
      result=true;
    }
```

```
    }

    return(result);
  }

  private boolean choosePrevious() {
    long now=System.currentTimeMillis();
    boolean result=false;

    if (lastUpKeyTime==-1 || now-lastUpKeyTime>KEY_TIME_DELTA) {
      lastUpKeyTime=now;
      lastDownKeyTime=-1L;

      int checked=choiceMode.getCheckedPosition();

      if (checked>0) {
        onChecked(checked-1, true, true);
        result=true;
      }
      else if (checked<0) {
        onChecked(0, true, true);
        result=true;
      }
    }

    return(result);
  }
```

(from RecyclerView/SingleActivatedListKB/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/ChoiceCapableAdapter.java)

In both cases, we find out what the current selection is. If it is a negative number, we do not have a selection yet, so we select the first row. Otherwise, if we can still move in the desired direction, add or subtract one from the current selection.

However, in this crude implementation, we need to slow down how frequently we change the selection. Simply changing which row is highlighted is fast, but if the list has to scroll to uncover that row, doing too many of those too quickly results in a "smearing" effect. The simplest way to avoid that smearing is to limit how many consecutive identical key events we process (e.g., user holds down an arrow key).

The approach taken in this code is to track the last time we were called to process an arrow key event for each direction, in `lastUpKeyTime` and `lastDownKeyTime` fields. We use `-1` to indicate that we have not just processed another one of that type. If, when we get a key event, either the related time value is `-1` or is within `KEY_TIME_DELTA` of now, we go ahead and update the checked position (if needed),

plus update the times. `KEY_TIME_DELTA` is defined as `250`, limiting us to four updates per second.

We change the current selection, where needed, via a call to a new three-parameter `onChecked()` method:

```
void onChecked(int position, boolean isChecked) {
  onChecked(position, isChecked, false);
}

void onChecked(int position, boolean isChecked, boolean updateUI) {
  if (choiceMode.isSingleChoice()) {
    int checked=choiceMode.getCheckedPosition();

    if (checked>=0) {
      RowController row=
          (RowController)rv.findViewHolderForAdapterPosition(checked);

      if (row!=null) {
        row.setChecked(false);
      }
    }
  }

  choiceMode.setChecked(position, isChecked);

  if (updateUI) {
    notifyItemChanged(position);
    rv.scrollToPosition(position);
  }
}
```

(from RecyclerView/SingleActivatedListKB/app/src/main/java/com/commonsware/android/recyclerview/singleactivatedlist/ChoiceCapableAdapter.java)

The third parameter indicates if we need to update the UI or not. For touchscreen events, activating rows and such is enough to ensure that the UI is properly updated. With key events, we need to:

- Make sure that Android is going to repaint the affected row and pick up the change in the activated state (`notifyItemChanged()`), and
- Make sure that the user can *see* the affected row, since they key event might now select a row that is not visible on the screen (`scrollToPosition()`)

Action Modes

Another thing that `ListView` gave us was support for action modes. In particular, the "multiple-choice modal" setting would automatically start and finish an action mode for us.

And, once again, `RecyclerView` has no hooks for action modes, though you can do it yourself if desired. We have to manually start and destroy the action mode, in addition to responding to the user's interaction with the action mode (tapping on items, or dismissing the action mode manually).

Where things get interesting is in the connection between checked items and the action mode. There are two UX rules:

1. When there are no checked items, there should be no action mode
2. When there is no action mode, there should be no checked items

You might think that those two rules are the same, and to some extent they are. They are phrased this way to emphasize the state changes that are involved:

- When the user checks an item, an action mode should appear
- When the user unchecks the last checked item, and therefore there are no more checked items, the action mode should disappear
- When the user dismisses the action mode (e.g., presses BACK), all checked items should become unchecked

Handling these transitions takes a bit of work, demonstrated in the `RecyclerView/ActionModeList` sample project. This is a clone of the `ChoiceList` sample from earlier, augmented with an action mode when 1+ items are checked. The action mode logic is largely cloned from one of the book's action mode samples, where we want to allow the user to capitalize or remove the checked items.

Once again, we have some tweaks to `ChoiceMode`, adding two methods:

1. `getCheckedCount()`, to return the number of checked items, which we will use for the subtitle of the action mode
2. `clearChecks()`, to uncheck all checked items

```
package com.commonsware.android.recyclerview.actionmodelist;

import android.os.Bundle;
```

```
public interface ChoiceMode {
  void setChecked(int position, boolean isChecked);
  boolean isChecked(int position);
  void onSaveInstanceState(Bundle state);
  void onRestoreInstanceState(Bundle state);
  int getCheckedCount();
  void clearChecks();
}
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/ChoiceMode.java)

`MultiChoiceMode` implements those, plus adds a subtle change to `setChecked()`. Previous editions of `MultiChoiceMode` would simply put the checked state `boolean` into the `ParceableSparseBooleanArray`, with `false` as a default value for any position not in the array. Now, we specifically *remove* items that are unchecked, so the only items in the `ParcelableSparseBooleanArray` are those that are checked. This makes `getCheckedCount()` and `clearChecks()` very simple to implement:

```
package com.commonsware.android.recyclerview.actionmodelist;

import android.os.Bundle;

public class MultiChoiceMode implements ChoiceMode {
  private static final String STATE_CHECK_STATES="checkStates";
  private ParcelableSparseBooleanArray checkStates=new ParcelableSparseBooleanArray();

  @Override
  public void setChecked(int position, boolean isChecked) {
    if (isChecked) {
      checkStates.put(position, isChecked);
    }
    else {
      checkStates.delete(position);
    }
  }

  @Override
  public boolean isChecked(int position) {
    return(checkStates.get(position, false));
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    state.putParcelable(STATE_CHECK_STATES, checkStates);
  }

  @Override
  public void onRestoreInstanceState(Bundle state) {
    checkStates=state.getParcelable(STATE_CHECK_STATES);
  }

  @Override
  public int getCheckedCount() {
    return(checkStates.size());
```

```
  }

  @Override
  public void clearChecks() {
    checkStates.clear();
  }
}
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/MultiChoiceMode.java)

`ChoiceCapableAdapter` exposes the two new `ChoiceMode` capabilities to its subclasses:

```
package com.commonsware.android.recyclerview.actionmodelist;

import android.os.Bundle;
import android.support.v7.widget.RecyclerView;

abstract public class
    ChoiceCapableAdapter<T extends RecyclerView.ViewHolder>
    extends RecyclerView.Adapter<T> {
  private final ChoiceMode choiceMode;

  public ChoiceCapableAdapter(ChoiceMode choiceMode) {
    super();
    this.choiceMode=choiceMode;
  }

  void onChecked(int position, boolean isChecked) {
    choiceMode.setChecked(position, isChecked);
  }

  boolean isChecked(int position) {
    return(choiceMode.isChecked(position));
  }

  void onSaveInstanceState(Bundle state) {
    choiceMode.onSaveInstanceState(state);
  }

  void onRestoreInstanceState(Bundle state) {
    choiceMode.onRestoreInstanceState(state);
  }

  int getCheckedCount() {
    return(choiceMode.getCheckedCount());
  }
```

```
  void clearChecks() {
    choiceMode.clearChecks();
  }
}
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/ChoiceCapableAdapter.java)

`IconicAdapter` now not only extends `ChoiceCapableAdapter`, but it implements the `ActionMode.Callback` interface, and therefore will be responsible for managing the action mode:

```
class IconicAdapter extends ChoiceCapableAdapter<RowController>
    implements ActionMode.Callback {
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/MainActivity.java)

`IconicAdapter` now overrides `onChecked()`, normally just handled by `ChoiceCapableAdapter`. In addition to chaining to the superclass for standard behavior, `IconicAdapter` manages the action mode:

- If we do not have any checked items, and we have an active action mode, `finish()` that action mode, as the user has unchecked the last checked item and the action mode is no longer needed
- Otherwise, update the subtitle, and if we are checking an item (`isChecked` is `true`), and if we do not already have an action mode going (tracked by an `activeMode` data member), start the action mode using `startActionMode()`

```
  @Override
  void onChecked(int position, boolean isChecked) {
    super.onChecked(position, isChecked);

    if (getCheckedCount()==0 && activeMode!=null) {
      activeMode.finish();
    }
    else {
      updateSubtitle(activeMode);

      if (isChecked) {
        if (activeMode==null) {
          activeMode=startActionMode(this);
        }
      }
    }
  }
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/MainActivity.java)

Because `IconicAdapter` implements `ActionMode.Callback`, it needs to implement the methods required by that interface. This includes:

- `onCreateActionMode()`, to set up the action mode
- `onPrepareActionMode()`, just because it is required by the interface
- `onActionItemClicked()`, where we should do some real work, but for the moment just have a `TODO` comment
- `onDestroyActionMode()`, where we make sure that all checked items are unchecked (`clearChecks()`) and tell the `RecyclerView.Adapter` that the data set changed, to force a repaint of all the visible rows, so they will now reflect the fact that they are no longer checked

```
  @Override
  public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    MenuInflater inflater=getMenuInflater();

    inflater.inflate(R.menu.context, menu);
    mode.setTitle(R.string.context_title);
    activeMode=mode;
    updateSubtitle(activeMode);

    return(true);
  }

  @Override
  public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
    return(false);
  }

  @Override
  public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    // TODO: do something based on the action

    updateSubtitle(activeMode);

    return(true);
  }

  @Override
  public void onDestroyActionMode(ActionMode mode) {
    if (activeMode != null) {
      activeMode=null;
      clearChecks();
      notifyDataSetChanged();
```

```
    }
  }
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/MainActivity.java)

The `updateSubtitle()` method, referred to by some of the previous methods, just updates the subtitle of the action mode to reflect the current count of checked items:

```
  private void updateSubtitle(ActionMode mode) {
    if (mode!=null) {
      mode.setSubtitle("("+getCheckedCount()+")");
    }
  }
```

(from RecyclerView/ActionModeList/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist/MainActivity.java)

The resulting app looks a lot like the original `ChoiceList` sample, until we check one or more items, at which point the action mode appears:

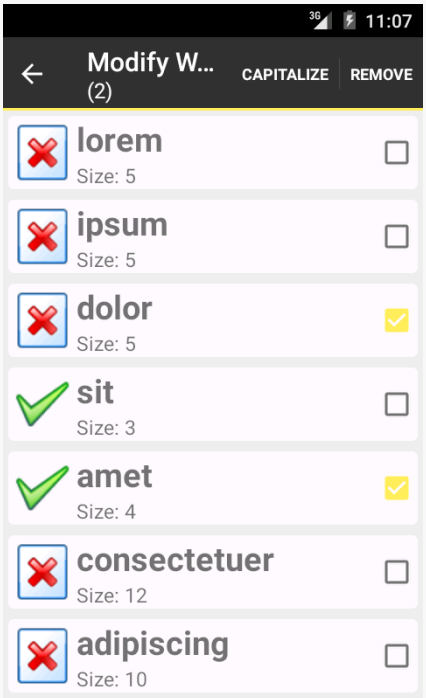


Figure 547: ActionModeList RecyclerView Demo

Changing the Contents

The obvious problem with the preceding sample is that we are not actually doing anything in response to user clicks on action mode items. We really should be capitalizing and/or removing words. However, this involves modifying the model data and how that model data is being visually displayed by the `RecyclerView`.

The less-obvious problem is that we are calling `notifyDataSetChanged()` when the action mode is dismissed, to force a full repaint of the `RecyclerView` contents. While this works, it is overkill, as probably only a subset of the visible items are checked. Ideally, we would only update the specific positions that were checked and now, with the action mode finished, are unchecked. We could find the affected `RowController` instances, using `findViewHolderByPosition()` on `RecyclerView`, as we did in the single-choice list sample. But, really, updating the checked state is just another manifestation of the same problem that capitalizing or removing words causes: we need to ensure that the `RecyclerView` depicts the current model state, ideally with minimum work.

So, let's see how this is accomplished, by looking at the `RecyclerView/ActionModeList2` sample project. As the name suggests, this is a clone of the `ActionModeList` shown in the preceding section. This time, we will fully implement `onActionItemClicked()` and allow our model data to be mutable.

Updating Existing Contents

With `AdapterView` and `Adapter` classes based on `BaseAdapter`, the only way we had to tell the `AdapterView` about model data changes was `notifyDataSetChanged()`. This would trigger a rebuild of the entire `AdapterView`, which is slow and expensive.

While `RecyclerView.Adapter` has its own `notifyDataSetChanged()`, that is really for total reloads of the model data, such as having gotten a fresh `Cursor` from a database and not knowing exactly what the changes are. If you are driving the changes yourself from the UI — and particularly if your model data is something like an `ArrayList` of model objects – you can use methods on `RecyclerView.Adapter` that are more fine-grained than is `notifyDataSetChanged()`.

If an item was updated in place — such as a word now being capitalized – you can use `notifyItemChanged()` on `RecyclerView.Adapter` to point out the specific position that changed. Alternatives include:

- `notifyItemMoved()`, to indicate that an item is still in the model data but now is in a new position
- `notifyItemRangeChanged()`, to indicate a range of positions that were modified, instead of having to repeatedly call `notifyItemChanged()`

`ActionModeList2` uses `notifyItemChanged()` when the user capitalizes words, to get those items repainted, if needed. It may not be needed immediately, if one or more of those items are not presently visible within the `RecyclerView`.

However, so far, our model data has been a static `String` array, and now we need a mutable model. So, we take the same approach as the `ListView` action mode samples use, converting our model to be an `ArrayList` that happens to be populated by a static `String` array.

The `items` data member of `MainActivity` is now an `ArrayList` of `String`, with the static `String` array being converted into `ORIGINAL_ITEMS`:

```
private static final String[] ORIGINAL_ITEMS={"lorem", "ipsum", "dolor",
        "sit", "amet",
        "consectetuer", "adipiscing", "elit", "morbi", "vel",
        "ligula", "vitae", "arcu", "aliquet", "mollis",
        "etiam", "vel", "erat", "placerat", "ante",
        "porttitor", "sodales", "pellentesque", "augue", "purus"};
private ArrayList<String> items;
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MainActivity.java)

Places that used to refer to `items` now use a private `getItems()` method, which lazy-instantiates the list if needed:

```
private ArrayList<String> getItems() {
  if (items==null) {
    items=new ArrayList<String>();

    for (String s : ORIGINAL_ITEMS) {
      items.add(s);
    }
  }

  return(items);
}
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MainActivity.java)

We also need to ensure that we hold onto the items across configuration changes, since those items could be changed by the user. So, our `onSaveInstanceState()` and `onRestoreInstanceState()` methods on `MainActivity` now handle that chore, in addition to their original behavior of having the `ChoiceCapableAdapter` persist checked states:

```
  @Override
  public void onSaveInstanceState(Bundle state) {
    adapter.onSaveInstanceState(state);
    state.putStringArrayList(STATE_ITEMS, items);
  }

  @Override
  public void onRestoreInstanceState(Bundle state) {
    adapter.onRestoreInstanceState(state);
    items=state.getStringArrayList(STATE_ITEMS);
  }
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MainActivity.java)

(here, `STATE_ITEMS` is a static data member, serving as the constant key for the `Bundle` entry)

In order to be able to capitalize or remove the checked words from the list, we need to know which ones are checked. Rather than expose that data directly, `ChoiceMode` now has a `visitChecks()` method, where we can supply a `Visitor` to be invoked for every checked position:

```
package com.commonsware.android.recyclerview.actionmodelist2;

import android.os.Bundle;

public interface ChoiceMode {
  void setChecked(int position, boolean isChecked);
  boolean isChecked(int position);
  void onSaveInstanceState(Bundle state);
  void onRestoreInstanceState(Bundle state);
  int getCheckedCount();
  void clearChecks();
  void visitChecks(Visitor v);

  public interface Visitor {
    void onCheckedPosition(int position);
  }
}
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/ChoiceMode.java)

`MultiChoiceMode` implements `visitChecks()` by iterating over a *copy* of the `checkStates` `ParcelableSparseBooleanArray`. That way, if the visitor modifies `checkStates` (e.g., unchecks a position), our loop is unaffected.

```
@Override
public void visitChecks(Visitor v) {
  SparseBooleanArray copy=checkStates.clone();

  for (int i=0;i<copy.size();i++) {
    v.onCheckedPosition(copy.keyAt(i));
  }
}
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MultiChoiceMode.java)

`visitChecks()` is also exposed by `ChoiceCapableAdapter`, as are all the other methods on `ChoiceMode`.

Now, `IconicAdapter` can capitalize the words, by using `visitChecks()`:

```
      case R.id.cap:
        visitChecks(new ChoiceMode.Visitor() {
          @Override
          public void onCheckedPosition(int position) {
            String word=getItems().get(position);

            word=word.toUpperCase(Locale.ENGLISH);
            getItems().set(position, word);
            notifyItemChanged(position);
          }
        });
        break;
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MainActivity.java)

Here, for each checked item, we capitalize the word, replace the original word with its capitalized equivalent, and call `notifyItemChanged()` to let the `RecyclerView` know that this position had its model data changed and therefore should be repainted, if needed.

We also use `visitChecks()` now in `onDestroyActionMode()`, to avoid the `notifyDataSetChanged()` call:

```
  @Override
  public void onDestroyActionMode(ActionMode mode) {
    if (activeMode != null) {
      activeMode=null;
      visitChecks(new ChoiceMode.Visitor() {
        @Override
        public void onCheckedPosition(int position) {
          onChecked(position, false);
          notifyItemChanged(position);
        }
      });
    }
  }
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MainActivity.java)

Each item that was checked is unchecked, and we use `notifyItemChanged()` to ensure that the item is repainted if needed.

Now, checking some items and choosing "CAPITALIZE" from the action mode will capitalize those words:

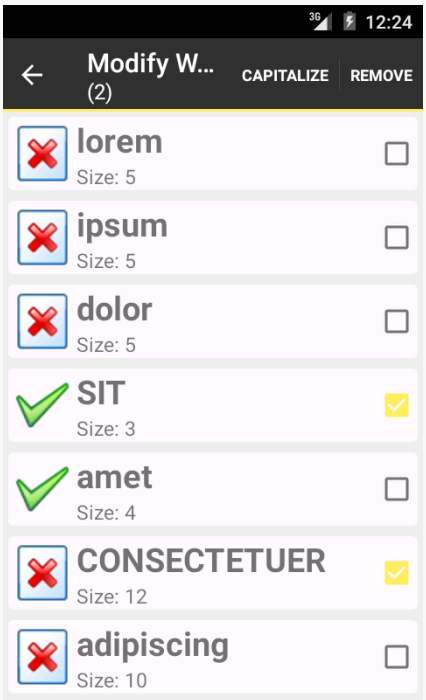


Figure 548: ActionModeList2 RecyclerView Demo, with Capitalized Words

Adding and Removing Items

There are also methods on `RecyclerView.Adapter` to specifically call out when you are adding or removing items from the adapter. Not only does this cause the `RecyclerView` to update itself, but it will animate the changes, if the relevant position(s) are visible.

Specifically, you can call:

- `notifyItemInserted()`, to indicate that a new item was inserted at a specified position, with everything else moving one position later in the roster
- `notifyItemRangeInserted()`, to insert several items in a block
- `notifyItemRemoved()`, to indicate a position that had an item removed from the roster, with later items moving up to take over earlier positions
- `notifyItemRangeRemoved()`, to remove several items in a block

The `ActionModeList2` sample uses `notifyItemRemoved()` as part of its handling of the `remove` action mode item:

```
        case R.id.remove:
          final ArrayList<Integer> positions=new ArrayList<Integer>();

          visitChecks(new ChoiceMode.Visitor() {
            @Override
            public void onCheckedPosition(int position) {
              positions.add(position);
            }
          });

          Collections.sort(positions, Collections.reverseOrder());

          for (int position : positions) {
            getItems().remove(position);
            notifyItemRemoved(position);
          }

          clearChecks();
          activeMode.finish();
          break;
```

(from RecyclerView/ActionModeList2/app/src/main/java/com/commonsware/android/recyclerview/actionmodelist2/MainActivity.java)

Because items slide up to take over vacated positions, when removing items, it is important to remove the lowest items first and work your way up the roster. That is why this code:

- Aggregates the list of positions that are checked
- Sorts the checked items in reverse order
- Iterates over the checked items, removing each from the `ArrayList` and calling `notifyItemRemoved()` to inform the adapter that the old item at this position is now gone
- Clears all of the checks from the `ChoiceMode` (as all checked items are now removed) and finishes the action mode (as there are no more checked items)

The result is that when the user removes items, they rapidly fade out, then later items in the list slide up to occupy the now-vacated space. If you would prefer to use other animations, you can do so, by creating your own subclass of `RecyclerView.ItemAnimator` and attaching it to the `RecyclerView` with `setItemAnimator()`.

The Order of Things

Version 22+ of `recyclerview-v7` offers `SortedList`. On the surface, the class appears to be a regular `List` that offers sorting. However, it also has a callback interface designed to be tied into `RecyclerView`, so that changes made to the `SortedList` can be reflected in the `RecyclerView` itself, complete with animations, optional batched processing, and so on.

This is illustrated in the `RecyclerView/SortedList` sample project. Along the way, we will also see how to use `RecyclerView` in a fragment and how to populate `RecyclerView` from the background thread.

The Gradle Change

This project requires version 22 or higher of `recyclerview-v7`, as the original v21 release of `recyclerview-v7` did not have `SortedList`. So, the Gradle build file requests something appropriate:

```
    implementation 'com.android.support:recyclerview-v7:22.2.0'
    implementation 'com.android.support:cardview-v7:22.2.0'
```

(from RecyclerView/SortedList/app/build.gradle)

Note that it pulls in the same version of `cardview-v7`. In general, it is best to try to keep Android Support package libraries in sync, at least in terms of major versions. Similarly, it is best to have the `compileSdkVersion` match the major library version, as the library may be conditionally using APIs made available in that version of Android. Hence, the project also has `compileSdkVersion` (and `buildTools`) set to pull from v22:

```
    compileSdkVersion 22
    buildToolsVersion '26.0.2'
```

(from RecyclerView/SortedList/app/build.gradle)

The RecyclerViewFragment

Prior samples in this chapter used a `RecyclerViewActivity` for basic `RecyclerView` setup. However, in this sample, we want to use a retained fragment for managing the `AsyncTask`, which suggests putting the `RecyclerView` in a fragment, rather than having it be managed directly by the activity.

So, this project has a reworking of `RecyclerViewActivity` into `RecyclerViewFragment`:

```
package com.commonsware.android.recyclerview.sorted;

import android.app.Fragment;
import android.os.Bundle;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class RecyclerViewFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
    RecyclerView rv=new RecyclerView(getActivity());

    rv.setHasFixedSize(true);

    return(rv);
  }

  public void setAdapter(RecyclerView.Adapter adapter) {
    getRecyclerView().setAdapter(adapter);
  }
```

```
  public RecyclerView.Adapter getAdapter() {
    return(getRecyclerView().getAdapter());
  }

  public void setLayoutManager(RecyclerView.LayoutManager mgr) {
    getRecyclerView().setLayoutManager(mgr);
  }

  public RecyclerView getRecyclerView() {
    return((RecyclerView)getView());
  }
}
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/RecyclerViewFragment.java)

Basically, what had been in `onCreate()` mostly moves into `onCreateView()`, where we set up the `RecyclerView`. The rest of the core API is unchanged.

The project has `SortedFragment`, which extends `RecyclerViewFragment` and handles loading of the data — we will examine more of it later in this chapter.

The revised `MainActivity` then just loads up `SortedFragment` via a `FragmentTransaction`:

```
package com.commonsware.android.recyclerview.sorted;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new SortedFragment()).commit();
    }
  }
}
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/MainActivity.java)

The SortedFragment

Most of `SortedFragment` is reminiscent of the original `AsyncTask` demo from the chapter on threads, mashed up with one of the `CardView`/`RecyclerView` samples from earlier in this chapter. However, the `SortedList` gets weaved throughout.

The SortedList

The `model` in the original `AsyncTask` demo was a simple `ArrayList`. Now it is a `SortedList`, initialized in `onCreate()` of the `SortedFragment`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  model=new SortedList<String>(String.class, sortCallback);

  task=new AddStringTask();
  task.execute();
}
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/SortedFragment.java)

The `SortedList` constructor takes two parameters:

- the Java class object for the models inside the list (in this case, `String.class`)
- a `SortedList.Callback` object that will be invoked when the model changes based on `List` APIs (e.g., `add()`, `insert()`, `remove()`)

There is an optional third parameter for the capacity, unused in this sample.

We will take a peek at the `SortedList.Callback` implementation, named `sortCallback`, shortly.

The IconicAdapter

The `IconicAdapter` from earlier `RecyclerView` samples worked directly off of the `static` array of `String` values. Now, we want it to work off of the `model` `SortedList`.

Hence, `onBindViewHolder()` and `getItemCount()` need to be modified to refer to appropriate methods on the `model`:

```
class IconicAdapter extends RecyclerView.Adapter<RowController> {
  @Override
  public RowController onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowController(getActivity().getLayoutInflater()
                             .inflate(R.layout.row, parent, false)));
  }

  @Override
  public void onBindViewHolder(RowController holder, int position) {
    holder.bindModel(model.get(position));
  }

  @Override
  public int getItemCount() {
    return(model.size());
  }
}
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/SortedFragment.java)

Also note that when we create the adapter in `onViewCreated()`, that we hold onto it in an `adapter` data member of the fragment:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  setLayoutManager(new LinearLayoutManager(getActivity()));
  adapter=new IconicAdapter();
  setAdapter(adapter);
}
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/SortedFragment.java)

The SortedList.Callback

The job of the `SortedList.Callback` is to serve as the bridge between the `SortedList` and the `RecyclerView.Adapter`.

`SortedList`, as the name suggests, sorts its contents. That means that any change to the `SortedList` contents can have different impacts on the `RecyclerView`. For example, while an `add()` to an `ArrayList` would just add a new row to the end of the

RecyclerView, an add() on SortedList might need to insert a row in the middle of the RecyclerView, to maintain the sorted order.

Hence, your SortedList.Callback is responsible for two things:

- Helping with the sorting itself, by comparing elements
- Passing information about how the sorting is done out to the RecyclerView.Adapter, so the appropriate moves can be made there, complete with animations

With that in mind, here is the sortedCB implementation of SortedList.Callback:

```
private SortedList.Callback<String> sortCallback=new SortedList.Callback<String>() {
  @Override
  public int compare(String o1, String o2) {
    return o1.compareTo(o2);
  }

  @Override
  public boolean areContentsTheSame(String oldItem, String newItem) {
    return(areItemsTheSame(oldItem, newItem));
  }

  @Override
  public boolean areItemsTheSame(String oldItem, String newItem) {
    return(compare(oldItem, newItem)==0);
  }

  @Override
  public void onInserted(int position, int count) {
    adapter.notifyItemRangeInserted(position, count);
  }

  @Override
  public void onRemoved(int position, int count) {
    adapter.notifyItemRangeRemoved(position, count);
  }

  @Override
  public void onMoved(int fromPosition, int toPosition) {
    adapter.notifyItemMoved(fromPosition, toPosition);
  }

  @Override
  public void onChanged(int position, int count) {
    adapter.notifyItemRangeChanged(position, count);
  }
};
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/SortedFragment.java)

The first method is your standard sort of compare() comparison method, as you might implement on a Comparator. It should return zero if the two model objects

are the same from a sorting standpoint, a negative number if the first parameter sorts before the second parameter, or a positive number if the first parameter sorts after the second parameter.

Then there are two similarly-named methods that serve as more-or-less replacements for the `equals()` that you might have on a `Comparator`: `areContentsTheSame()` and `areItemsTheSame()`.

`areItemsTheSame()` should return `true` if the two passed-in values represent the same actual logical item. In the case of `SortedFragment`, that is simply whether or not the strings are equal. But, with a more complex data model, you might be comparing primary keys or some other form of immutable identifier.

`areContentsTheSame()` should return `true` if the visual representation of the items look the same, as this will be used to optimize the changes made to the `RecyclerView`.

For example, suppose a shopping cart fragment wanted to use `SortedList`. Further suppose that if you added three boxes of laundry detergent to the cart, rather than having one row in the list with "Quantity: 3", you were representing them as three rows in the `RecyclerView`. In this case:

- `compare()` returns a value to indicate the sorting rules of those shopping cart items, perhaps based on the title of the item
- `areItemsTheSame()` might return `false` for any combination of these three items, as they are logically distinct rows within the `RecyclerView`
- `areContentsTheSame()` might return `true` for any combination of these three items, as while they are three separate line items, each is visually identical in terms of what the `RecyclerView` rows look like

In many cases, `areContentsTheSame()` can simply invoke `areItemsTheSame()`, under the premise that different items *probably* have different visual representations. That is what is done in this sample, where `areItemsTheSame()` in turn uses `compare()` to see whether or not the items are the same.

Finally, there are four `on...()` methods that are simply forwarded along to their `RecyclerView.Adapter` counterparts, so changes to the `SortedList` make the corresponding changes to the `RecyclerView` contents.

Note that there is a `SortedListAdapterCallback` that takes a `RecyclerView.Adapter` as a constructor parameter and handles the `on...()` methods for you. However,

since we want to retain the `SortedList` across configuration changes, and since `SortedList` does not allow us to change the `SortedList.Callback` object, we cannot readily switch the `SortedList` to the new fragment and new adapter after a configuration change.

The AsyncTask

The `AddStringTask` is the same as with the original `AsyncTask` sample, except that now it adds the words to the `SortedList`, which (via its `Callback`) will update the `RecyclerView`:

```
private class AddStringTask extends AsyncTask<Void, String, Void> {
  @Override
  protected Void doInBackground(Void... unused) {
    for (String item : items) {
      if (isCancelled())
        break;

      publishProgress(item);
      SystemClock.sleep(400);
    }

    return(null);
  }

  @Override
  protected void onProgressUpdate(String... item) {
    if (!isCancelled()) {
      model.add(item[0]);
    }
  }

  @Override
  protected void onPostExecute(Void unused) {
    Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
        .show();

    task=null;
  }
}
```

(from RecyclerView/SortedList/app/src/main/java/com/commonsware/android/recyclerview/sorted/SortedFragment.java)

The Results

If you run this sample, you will see the words be added to the list, every 400ms. However, in the original `ListView`-based sample, new rows were appended to the end, and so you would not see new rows appear after the `ListView` space was filled. In this sample, the Latin words are sorted by `SortedList`, and you will see them animate into position at the appropriate spots as they are added. In the end, you get the same look as in earlier `CardView`-based `RecyclerView` implementations, except that the words are sorted:

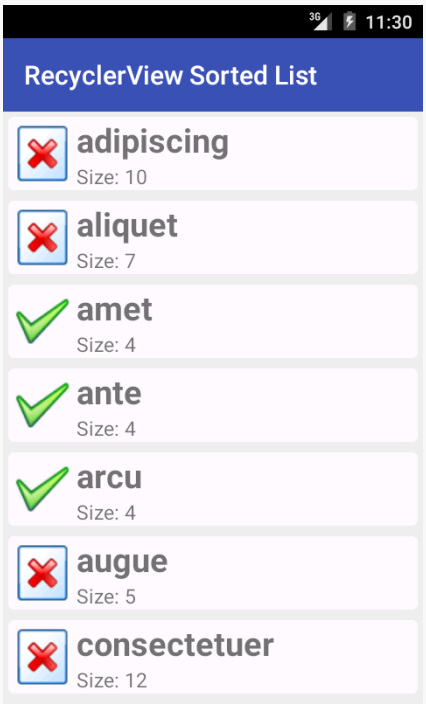


Figure 549: SortedList RecyclerView Demo

Other Bits of Goodness

To quote the infamous American infomercial line: "But wait! There's more!"

In addition to `LinearLayoutManager` and `GridLayoutManager`, there is `StaggeredGridLayoutManager`. With a vertically-scrolling `GridLayoutManager`, rows are all a consistent height, but the cell widths might vary. With a vertically-scrolling `StaggeredGridLayoutManager`, the columns are all the same width, but the cell heights might vary.

All three of the standard layout managers support horizontal operation as well, through a `boolean` on a constructor. In these cases, the content will scroll horizontally, rather than vertically. This eliminates the need for third-party horizontal `ListView` implementations and the like.

And, of course, you can implement your own `RecyclerView.LayoutManager`, avoiding any of the built-in ones.

Animating the Deltas Using DiffUtil

`SortedList` is interesting, but it is inflexible. For example, you cannot readily change the sort order, without completely replacing the `SortedList`.

Moreover, it assumes that the changes that you want to make are simply to keep a list in sorted order. There are plenty of other possible changes to your data set that might occur, such as from the results of some Web service call to synchronize your local data with that on a server. You would have to somehow perform your own "diff" on the data shown in your `RecyclerView` and the new roster of data, to determine what changed, what did not change, and how those changes affect things like item positions within the list or grid.

Fortunately, the 25.0.0+ version of `recyclerview-v7` gives you another tool: `DiffUtil`. This handles everything cited in the last sentence of the previous paragraph. You hand it two collections (old and new), plus an object that can help determine what the changes are (one reminiscent of a `SortedList.Callback`). It gives you a results object that, in turn, can update the `RecyclerView` to affect those changes.

The `Java8/VideoLambda` sample project is yet another rendition of the "list of videos" sample app from earlier in this chapter. However, it shows the list in sorted order, but using `DiffUtil` rather than `SortedList`. This allows us to offer the user the ability to change the sort order (ascending or descending).

Model

The original version of this sample used a `Cursor` directly, wrapping it in a `RecyclerView.Adapter`. That is fine, but we cannot readily sort a `Cursor`. It is simpler to convert the `Cursor` into a list of model objects, so we can sort the list.

To that end, the `VideoLambda` sample app has a `Video` class, with a constructor that can populate itself from a `Cursor` positioned on a valid row:

```
package com.commonsware.android.recyclerview.videolist;

import android.content.ContentUris;
import android.database.Cursor;
import android.net.Uri;
import android.provider.MediaStore;

class Video implements Comparable<Video> {
  final String title;
  final Uri videoUri;
  final String mimeType;

  Video(Cursor row) {
    this.title=
      row.getString(row.getColumnIndex(MediaStore.Video.Media.TITLE));
    this.videoUri=ContentUris.withAppendedId(
      MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
      row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID)));
    this.mimeType=
      row.getString(row.getColumnIndex(MediaStore.Video.Media.MIME_TYPE));
  }

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Video)) {
      return(false);
    }

    return(videoUri.equals(((Video)obj).videoUri));
  }

  @Override
  public int hashCode() {
    return(videoUri.hashCode());
  }

  @Override
  public int compareTo(Video video) {
    return(title.compareTo(video.title));
  }
}
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/Video.java)

Note that `Video` has an implementation of `equals()` considering two `Video` objects to be equal if they point to the same `Uri`. That will be important when we use `DiffUtil`, as `DiffUtil` (and our helper code) need to know when two `Video` objects logically represent the same video.

`Video` also implements `Comparable` and therefore has a `compareTo()` method, implemented by comparing the titles of the videos. This will be used as part of our sorting logic.

The `VideoAdapter` has a `setVideos()` method, taking a `Cursor` that we loaded from the `MediaStore`, as before. However, now `VideoAdapter` does not hold that `Cursor` in a field. Rather, it holds an `ArrayList` of `Video` objects as the `videos` field, with `setVideos()` handling the conversion of data from the `Cursor` into the `Video` objects:

```
  void setVideos(Cursor c) {
    if (c==null) {
      videos=null;
      notifyDataSetChanged();
    }
    else {
      ArrayList<Video> temp=new ArrayList<>();

      while (c.moveToNext()) {
        temp.add(new Video(c));
      }

      if (videos==null) {
        videos=new ArrayList<>();
      }

      sortAndApply(temp);
    }
  }

  @Override
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

We will see the `sortAndApply()` method, and understand what the `temp` local variable is all about, a bit later in this section. For now, take it on faith that:

- If the `Cursor` is `null` (e.g., `onLoaderReset()` was called), we `null` out the `videos` field and call `notifyDataSetChanged()`, to let the `RecyclerView` know that our entire roster of videos changed

- If the `Cursor` is not `null`, we convert it into an `ArrayList` of `Video` objects, then use `sortAndApply()` to update the `RecyclerView`

The Menu

The other time we need to sort the videos is if the user chooses to switch from ascending to descending sort (or back again).

The app has a menu resource with a `checkable` menu `<item>`, named `sort`, that the user will be able to use to toggle the sort order:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/sort"
    android:checkable="true"
    android:checked="true"
    android:enabled="false"
    android:showAsAction="never"
    android:title="@string/sort_ascending" />
</menu>
```

(from Java8/VideoLambda/app/src/main/res/menu/actions.xml)

Note that it is disabled initially. When the app starts up, we do not yet have our videos, since they need to be loaded from the `MediaStore`. Hence, we keep the item disabled until the videos are ready.

In the activity's `onCreateOptionsMenu()` method, we inflate the resource, get the `sort` `MenuItem`, hold onto it in a field (also named `sort`), and enable it if the `VideoAdapter` happens to already have some videos:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);
    sort=menu.findItem(R.id.sort);
    sort.setEnabled(adapter.getItemCount()>0);

    return(super.onCreateOptionsMenu(menu));
  }
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

Then, in `onOptionsItemSelected()`, if the user taps on this action bar item, we toggle its checked state, then tell the `VideoAdapter` to `sort()` based upon that state:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.sort) {
      item.setChecked(!item.isChecked());
      adapter.sort(item.isChecked());

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

`sort()`, in turn, keeps track of the current sort order, then calls the same `sortAndApply()` that we did in `setVideos()`:

```
        sortAscending=checked;
        sortAndApply(new ArrayList<>(videos));
      }
    }
  }
}
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

The Diff-ing

The job of `sortAndApply()` is to do what the name suggests: sort the videos and apply the sorted list to the `RecyclerView`. In principle, there are two possible scenarios:

1. We are showing the videos for the very first time
2. We are showing the videos again, after a re-sort, or perhaps after a new video was scanned by the `MediaStore`, triggering the `Loader` framework to hand us a fresh `Cursor`

Fortunately, we can handle both the same way:

```
        Collections.sort(newVideos,
          (one, two) -> one.compareTo(two));
      }
      else {
        Collections.sort(newVideos,
          (one, two) -> two.compareTo(one));
```

```
    }

    DiffUtil.Callback cb=new SimpleCallback<>(videos, newVideos);
    DiffUtil.DiffResult result=DiffUtil.calculateDiff(cb, true);

    videos=newVideos;
    result.dispatchUpdatesTo(this);
  }

  private void sort(boolean checked) {
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/MainActivity.java)

The new list of videos is the `newVideos` parameter to `sortAndApply()`. Based on the `sortAscending` value, we use `Collections.sort()` to sort the `newVideos` list... using a Java 8 lambda expression. Lambda expressions are covered elsewhere in the book. However, we could just sort using an anonymous inner class implementation of `Comparator`, such as this for sorting in ascending order:

```
Collections.sort(temp, new Comparator<Video>() {
  @Override
  public int compare(Video one, Video two) {
    return(one.compareTo(two));
  }
});
```

Given the newly-sorted list, we create an instance of a `DiffUtil.Callback` object, called `SimpleCallback`. We will see its implementation shortly. Its role is to help `DiffUtil` calculate the nature of the changes to list of videos, comparing what is in the `RecyclerView` now (`videos`) with what we want the `RecyclerView` to show (`newVideos`).

That `DiffUtil.Callback` object is passed to the `calculateDiff()` method on `DiffUtil`. The second parameter — `true` — indicates whether or not objects in the list may have moved. Sometimes, we know that the updates do not move objects around in the list, but merely insert new ones or remove existing ones. In such cases, `calculateDiff()` can optimize its tracking algorithm to run more efficiently. In our case, we most definitely are changing the positions of existing objects, and therefore we need to pass `true`.

In principle, `calculateDiff()` should be called on a background thread. For long lists and complicated comparisons, `calculateDiff()` could take long enough that you might exhibit some jank by taking up too much time on the main application

thread. In this case, the comparisons are cheap, and hopefully you do not have too many videos on your test device.

The result of `calculateDiff()` is a `DiffUtil.DiffResult` object. After updating our `video` field to be the sorted `newVideos` collection, we call `dispatchUpdatesTo()` on the `DiffResult`, to have it apply the changes to the `VideoAdapter`. We need to update `videos` first because `dispatchUpdatesTo()` may trigger fresh `onBindViewHolder()` calls, and we need to make sure that we are using the newly-sorted list for those.

The SimpleCallback

`SimpleCallback`, as the name suggests, is a naive implementation of `DiffUtil.Callback`:

```
package com.commonsware.android.recyclerview.videolist;

import android.support.v7.util.DiffUtil;
import java.util.ArrayList;

class SimpleCallback<T extends Comparable> extends DiffUtil.Callback {
  private final ArrayList<T> oldItems;
  private final ArrayList<T> newItems;

  public SimpleCallback(ArrayList<T> oldItems,
                        ArrayList<T> newItems) {
    this.oldItems=oldItems;
    this.newItems=newItems;
  }

  @Override
  public int getOldListSize() {
    return(oldItems.size());
  }

  @Override
  public int getNewListSize() {
    return(newItems.size());
  }

  @Override
  public boolean areItemsTheSame(int oldItemPosition,
                                 int newItemPosition) {
    return(oldItems.get(oldItemPosition)
      .equals(newItems.get(newItemPosition)));
```

```
  }

  @Override
  public boolean areContentsTheSame(int oldItemPosition,
                                    int newItemPosition) {
    return(oldItems.get(oldItemPosition)
      .compareTo(newItems.get(newItemPosition))==0);
  }
}
```

(from Java8/VideoLambda/app/src/main/java/com/commonsware/android/recyclerview/videolist/SimpleCallback.java)

Any `DiffUtil.Callback` needs to implement four key `abstract` methods:

- `getOldListSize()` and `getNewListSize()`, which return pretty much what their names would indicate
- `areItemsTheSame()`, where you need to indicate if a particular item from the old list (identified by position) represents the same logical entity as does a particular item from the new list (also identified by position)
- `areContentsTheSame()`, where you need to indicate if objects from the old and new list (identified by positions) are similar enough that the user would not notice a visual difference

Those latter two methods basically fill the same roles as do methods of the same names on a `SortedList.Callback`.

In the case of `SimpleCallback`, `areItemsTheSame()` uses `equals()`. Since `Video` implements `equals()` to compare the video `Uri` values, `areItemsTheSame()` will return `true` if the two `Video` objects point to the same video.

`areContentsTheSame()` leverages `compareTo()`, which compares the titles of the videos. If `compareTo()` return 0, the titles are the same, and so `areContentsTheSame()` returns `true` to indicate that the visual representation of the videos is the same. This might not actually be the case, as we are showing the video thumbnails in the rows, and so it is possible that we have two videos that have the same title but different thumbnails. In that case, `DiffUtil` might not cause `RecyclerView` to redraw one or the other row. An alternative approach would be to have `areContentsTheSame()` simply return the value of `areItemsTheSame()`. This covers the thumbnails issue, at the expense of possibly doing some unnecessary shuffling of `RecyclerView` items, for cases where we have two videos with identical titles and thumbnails. It will be up to you, in your own app, to determine the best implementation of `areContentsTheSame()` based on your UI.

The March of the Libraries

By this point in time, you may be wailing in anguish and rending your garments over how much is involved in getting `RecyclerView` going.

(pro tip: do not rend your garments in public, to avoid running afoul of indecency laws)

There is little doubt that `RecyclerView` is the epitome of "some assembly required". However, other developers have come to the forefront with libraries that can help fill in these gaps without you having to roll all the code yourself.

Note that the author has not tried many of these libraries, and listing them here neither is an endorsement of these libraries nor a knock on any libraries not listed here.

DynamicRecyclerView

The DynamicRecyclerView library offers:

- Drag and drop reordering of items, via a simple custom `OnItemTouchListener` implementation
- An implementation of the horizontal "swipe-to-dismiss" UI pattern, also via a custom `OnItemTouchListener` implementation
- An implementation of choice modes, akin to those shown in this chapter
- An implementation of click-style events, by handling them as touch events using yet another custom `OnItemTouchListener` implementation

Advanced RecyclerView

The Advanced RecyclerView library offers its own drag-and-drop and swipe-to-dismiss implementations. Rather than using `OnItemTouchListener` implementations, you implement certain interfaces on your `RecyclerView.Adapter` and `RecyclerView.ViewHolder` classes to support drag-and-drop and/or swipe-to-dismiss, plus work with "manager" classes to tie the support together.

It also supports an expandable item, where clicking on the item expands or collapses a set of views vertically beneath the item, offering an approach for using `RecyclerView` to replace `ExpandableListView`. And, it has a few item decorators, including a basic divider implementation.

SuperRecyclerView

The SuperRecyclerView library has its own swipe-to-dismiss implementation. It also supports the "swipe layout" pattern, where a horizontal swipe gesture slides out the main view and uncovers a set of contextual operations on the item.

It also offers:

- scrollbars (which are not enabled on `RecyclerView`)
- progress bars and/or empty views to handle the asynchronous loading of data
- an "endless" or "infinite scrolling" implementation, where when the user scrolls to the bottom of the data, you get a chance to go load more data
- sticky headers, where as the user scrolls, the top-most header remains pinned to the top of the `RecyclerView`, until replaced by a new header that scrolls to the top

However, this library requires that you inherit from a custom `RecyclerView` subclass.

FlexibleDivider

The FlexibleDivider library does just one thing: provides dividers. However, it offers *deep* support for dividers, where you can easily control all sorts of aspects, from color and width to margins and path effects (e.g., dashed lines versus solid lines).

The `RecyclerView/FlexDividerList` sample project is a clone of the `ManualDividerList` sample from earlier in the chapter, where the dividers are now provided by the FlexibleDivider library, which is loaded via the `build.gradle` file:

```
dependencies {
    implementation 'com.android.support:recyclerview-v7:22.2.0'
    implementation 'com.yqritc:recyclerview-flexibledivider:1.0.1'
}
```

(from RecyclerView/FlexDividerList/app/build.gradle)

Then, we use the library's `HorizontalDividerItemDecoration` to set up our dividers:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setLayoutManager(new LinearLayoutManager(this));
```

```
  RecyclerView.ItemDecoration decor=
      new HorizontalDividerItemDecoration.Builder(this)
        .color(getResources().getColor(R.color.primary))
        .build();

  getRecyclerView().addItemDecoration(decor);
  setAdapter(new IconicAdapter());
}
```

(from RecyclerView/FlexDividerList/app/src/main/java/com/commonsware/android/recyclerview/flexdivider/MainActivity.java)

The results are a solid blue divider:

Figure 550: FlexDividerList RecyclerView Demo

Of course, through the library, you can change a lot more about the divider than just its color, through its builder-style API.

Expandable Rows

A common pattern in vertically-scrolling lists is to have the rows expand and contract when clicked. This allows you to have more information inline in the list, without always taking up all of the vertical space that the information might require.

This might not sound very hard: just toggle the visibility of some widgets, perhaps using an animation.

However, what we really want is that when a row is expanded, that the entirety of the expanded row is visible, assuming that there is sufficient screen space for it. Otherwise, if the user happens to expand some row at the bottom of the list, the user might not realize that more information is available off the bottom of the screen.

Making this work requires knowing where the row is in the list, how much space will be required when it is expanded, whether the expanded row will fit given the `RecyclerView` size, and how to scroll the `RecyclerView` to make the row fit if needed.

That sounds complicated.

And so, we turn to a library: the `ExpandableLayout` provided by `com.github.SilenceDut:ExpandableLayout`. This library is demonstrated in the `RecyclerView/ExpandableRow` sample project. However, this sample project makes extensive use of the data binding framework, so you may wish to read that chapter before continuing with this section.

This app is another "list the recent questions on Stack Overflow" apps that have been profiled elsewhere in the book, starting with the chapter on Internet access. In this case, we are using a `RecyclerView` for the list, with the data binding framework populating the rows.

A Stack Overflow question has lots of possible pieces of data, far more than we would want to display in a `RecyclerView` row. Even showing a subset of this information would make for a really long list, as each row would be fairly large. So, instead, we will use `ExpandableLayout` to show the title, owner's avatar, and question score all the time and show the tags, view count, and answer count after the user taps on a row. Since we are "stealing" the click event to expand and collapse the row, we cannot use it for anything else, so we also want a "View" button in the expanded area, to allow the user to view the Stack Overflow question on the Stack Overflow Web site.

Our `build.gradle` file pulls in libraries for accessing the Stack Exchange API (Retrofit and Picasso), libraries for displaying the results (`recyclerview-v7` and `cardview-v7`), plus `ExpandableLayout`:

```
apply plugin: 'com.android.application'

repositories {
    maven { url "https://jitpack.io" }
}

dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit:retrofit:1.9.0'
    implementation 'com.android.support:recyclerview-v7:24.1.1'
    implementation 'com.android.support:cardview-v7:24.1.1'
    implementation 'com.github.SilenceDut:ExpandableLayout:1.2.0'
}

android {
    compileSdkVersion 24
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
        applicationId "com.commonsware.android.databind.expandable"
    }

    dataBinding {
        enabled = true
    }
}
```

(from RecyclerView/ExpandableRow/app/build.gradle)

An `ExpandableLayout` contains two children. The first child will be shown all the time, while the second child represents the additional content to be shown when the `ExpandableLayout` is expanded. So, our `res/layout/row.xml` resource contains an `ExpandableLayout`, wrapping around the desired UI. The `ExpandableLayout` itself is wrapped in a `CardView` for formatting, and the whole thing is inside a `<layout>` for the data binding framework:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <data>

    <import type="android.text.Html" />
```

```
  <import type="android.text.TextUtils" />

  <variable
    name="question"
    type="com.commonsware.android.databind.basic.Question" />

  <variable
    name="controller"
    type="com.commonsware.android.databind.basic.QuestionController" />
</data>

<android.support.v7.widget.CardView
  xmlns:cardview="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="4dp"
  cardview:cardCornerRadius="4dp">

  <com.silencedut.expandablelayout.ExpandableLayout
    android:id="@+id/row_content"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/selectableItemBackground"
    android:onTouch="@{controller::onTouch}"
    app:expWithParentScroll="true">

    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal">

      <ImageView
        android:id="@+id/icon"
        android:layout_width="@dimen/icon"
        android:layout_height="@dimen/icon"
        android:layout_gravity="center_vertical"
        android:contentDescription="@string/icon"
        android:padding="8dip"
        app:error="@{@drawable/owner_error}"
        app:imageUrl="@{question.owner.profileImage}"
        app:placeholder="@{@drawable/owner_placeholder}" />

      <TextView
        android:id="@+id/title"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"
        android:layout_weight="1"
        android:text="@{Html.fromHtml(question.title)}"
        android:textSize="20sp" />

      <TextView
        android:id="@+id/score"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:text="@{Integer.toString(question.score)}"
```

```
          android:textSize="40sp"
          android:textStyle="bold" />

      </LinearLayout>

      <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="8dp">

        <LinearLayout
          android:layout_width="0dp"
          android:layout_height="wrap_content"
          android:layout_weight="1"
          android:orientation="vertical">

          <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:ellipsize="end"
            android:maxLines="1"
            android:text='@{@string/tags+" "+TextUtils.join(", ", question.tags)}' />

          <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text='@{@string/views+" "+question.viewCount}' />

          <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text='@{@string/answers+" "+question.answerCount}' />

        </LinearLayout>

        <Button
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:onClick="@{()->controller.showQuestion(context, question)}"
          android:text="@string/btn_view" />
      </LinearLayout>
    </com.silencedut.expandablelayout.ExpandableLayout>
  </android.support.v7.widget.CardView>
</layout>
```

(from RecyclerView/ExpandableRow/app/src/main/res/layout/row.xml)

We do not need to do anything to teach `ExpandableLayout` to expand and collapse based upon click events, as that is built into the class. However, we do opt into one optional feature, via `app:expWithParentScroll="true"`. This indicates that we want to scroll the parent (here, referring to the `RecyclerView`) to allow the expanded `ExpandableLayout` to be fully visible where possible.

When the activity is launched, the rows are collapsed:

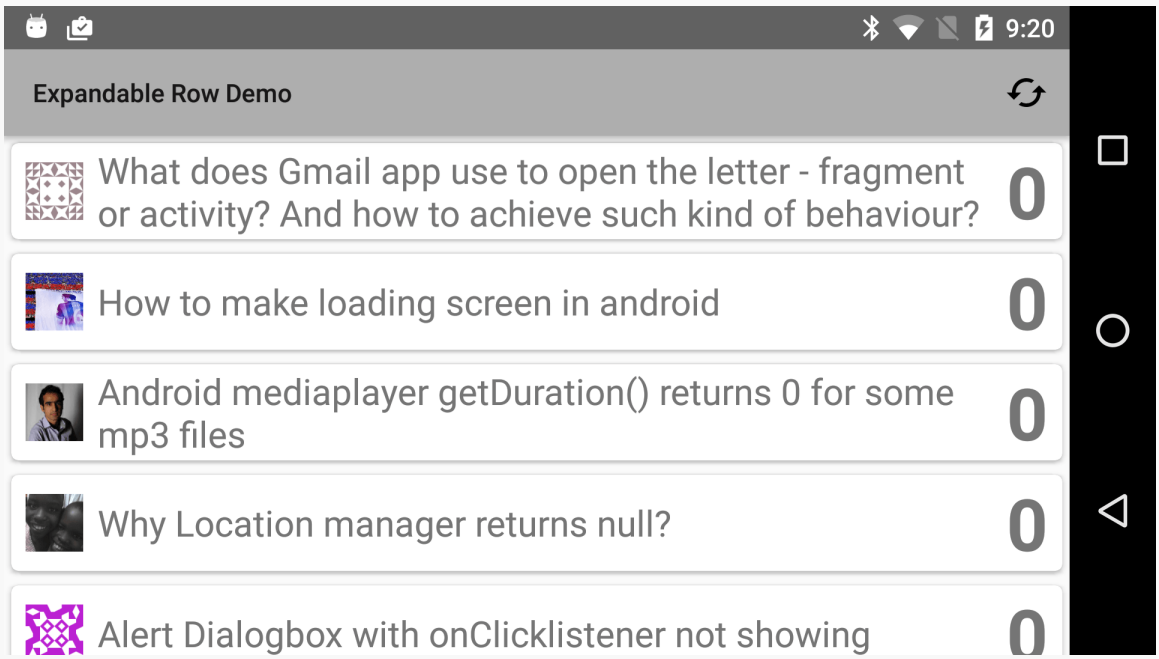
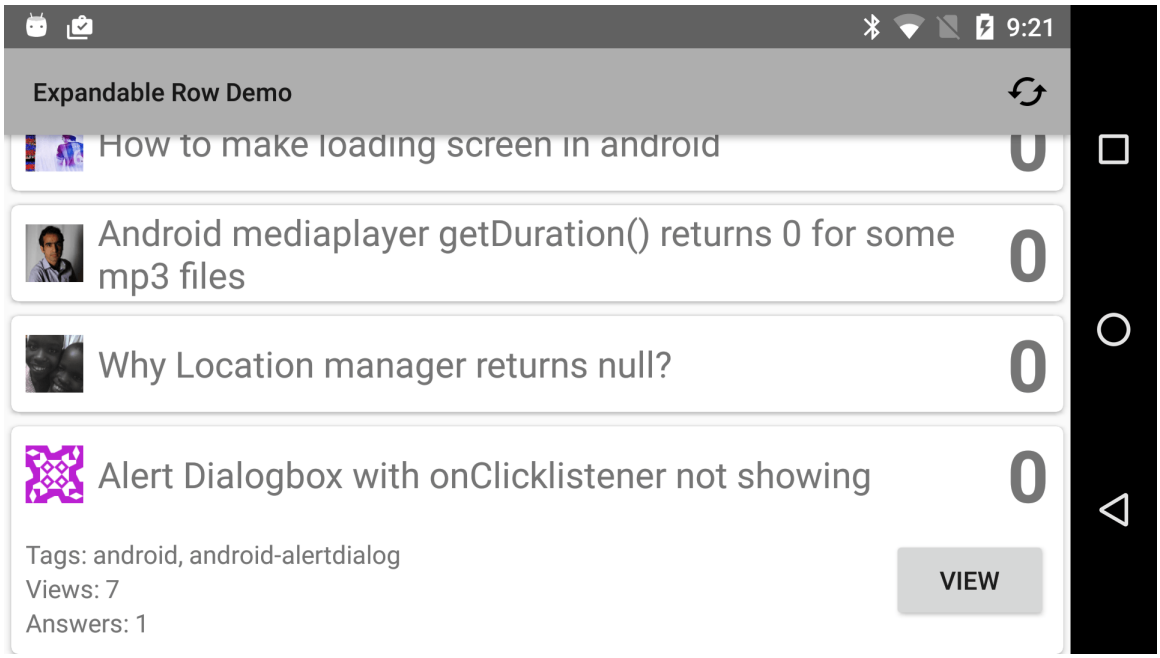


Figure 551: Expandable RecyclerView Rows, Collapsed

Tapping on one expands it to show the rest of the content, including scrolling the `RecyclerView` so it is visible:



Advanced RecyclerView

`RecyclerView` is the "Swiss army knife" of Android selection widgets. You can use it for a wide range of scenarios, well beyond what classic `AdapterView` widgets — like `ListView` or `GridView` — could handle.

In this chapter, we will "go outside the (`AdapterView`) box" and explore some advanced uses of `RecyclerView`.

Prerequisites

Understanding this chapter requires that you have read the preceding chapter, on `RecyclerView`.

RecyclerView as Pager

`ViewPager` has been used for horizontally-swiped page-at-a-time user interfaces since its debut in 2011.

However, `ViewPager` is not that flexible:

- You can only swipe horizontally. It has no `setSwipeDirection()` or similar method to switch to vertical swiping.
- It was designed to work with fragments as pages. While `PagerAdapter` itself can work with views as pages, even the minimum required API is difficult to understand. And using fragments as pages means that you may wind up in cases with nested fragments, which adds to the complexity.
- While `PagerAdapter` has enough hooks to allow you to add and remove pages on the fly, neither `FragmentPagerAdapter` nor

FragmentStatePagerAdapter support this use case, requiring you to roll your own PagerAdapter implementation.

And so on.

However, as it turns out, RecyclerView can be readily adapted to serve as a ViewPager replacement. Instead of a PagerAdapter, you use an ordinary RecyclerView.Adapter, where your pages are simple views. RecyclerView itself is far more flexible than is ViewPager, giving you a stronger foundation for more advanced paging scenarios.

Using RecyclerViewPager

The original solution for using RecyclerView as a ViewPager replacement came in the form of a third-party library, com.github.lsjwzh.RecyclerViewPager. This library offers a RecyclerViewPager subclass of RecyclerView that offers the page-at-a-time swiping metaphor.

The RecyclerViewPager/PlainRVP sample project illustrates its use. This is another rendition of the 10-EditText-widgets pager that was used in the chapter on ViewPager, swapping in RecyclerViewPager for the ViewPager.

Adding the Dependency

The com.github.lsjwzh.RecyclerViewPager library is not on JCenter or Maven Central. Instead, it is on jitpack.io, an artifact repository that builds artifacts directly from GitHub source repositories. So, to use this library, we need to add jitpack.io as a repository, in addition to adding the RecyclerViewPager library itself:

```
apply plugin: 'com.android.application'

repositories {
  maven { url "https://jitpack.io" }
}

dependencies {
    implementation 'com.android.support:recyclerview-v7:25.1.0'
    implementation 'com.github.lsjwzh.RecyclerViewPager:lib:v1.1.2'
}

android {
```

```
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 25
    }
}
```

(from RecyclerViewPager/PlainRVP/app/build.gradle)

Using the Widget

In the equivalent `ViewPager` sample app, the `main.xml` layout resource held the `ViewPager`. In this sample, it holds the `RecyclerViewPager` (or, more accurately, the `com.lsjwzh.widget.recyclerviewpager.RecyclerViewPager`, since the class name needs to be fully-qualified since it is coming from a library):

```
<?xml version="1.0" encoding="utf-8"?>
<com.lsjwzh.widget.recyclerviewpager.RecyclerViewPager android:id="@+id/pager"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:clipToPadding="false"
  android:layout_margin="@dimen/pager_padding"
  app:rvp_singlePageFling="true"
  app:rvp_triggerOffset="0.1" />
```

(from RecyclerViewPager/PlainRVP/app/src/main/res/layout/main.xml)

The `app:rvp_singlePageFling` indicates that we want to limit the user to switch one page at a time, rather than a long fling gesture resulting in moving through many pages at once. The `app:rvp_triggerOffset` attribute is undocumented but appears to control how much of a swipe gesture is necessary to trigger a page change.

Populating the Pages

With `ViewPager`, you supply the pages via a `PagerAdapter`, typically a `FragmentPagerAdapter` or a `FragmentStatePagerAdapter`. With `RecyclerViewPager`, you supply the pages via a `RecyclerView.Adapter`, just as you would with any other `RecyclerView`.

So, in `onCreate()` of the `MainActivity`, we get the `RecyclerViewPager`, hand it a horizontal `LinearLayoutManager`, create a `PageAdapter`, and attach that `PageAdapter` to the `RecyclerViewPager`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    RecyclerViewPager pager=(RecyclerViewPager)findViewById(R.id.pager);

    pager.setLayoutManager(new LinearLayoutManager(this,
      LinearLayoutManager.HORIZONTAL, false));
    adapter=new PageAdapter(pager, getLayoutInflater());
    pager.setAdapter(adapter);
  }
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

By using a horizontal `LinearLayoutManager`, the `RecyclerViewPager` will behave akin to a regular `ViewPager`, with navigation occurring via horizontal swipes. Want a vertical `ViewPager`? Replace the horizontal `LinearLayoutManager` with a vertical one, and you are set.

Our `PageAdapter` is a `RecyclerView.Adapter`, for a `RecyclerView.ViewHolder` named `PageController`. The basic setup for `PageAdapter` is not that different than any other `RecyclerView.Adapter`:

- We create a `PageController` in `onCreateViewHolder()`
- We populate that `PageController` with model data in `onBindViewHolder()`
- We indicate how many items there are in `getItemCount()`

```
class PageAdapter extends RecyclerView.Adapter<PageController> {
  private static final String STATE_BUFFERS="buffers";
  private static final int PAGE_COUNT=10;
  private final RecyclerViewPager pager;
  private final LayoutInflater inflater;
  private ArrayList<String> buffers=new ArrayList<>();

  PageAdapter(RecyclerViewPager pager, LayoutInflater inflater) {
    this.pager=pager;
    this.inflater=inflater;

    for (int i=0;i<10;i++) {
      buffers.add("");
    }
  }

  @Override
```

```
  public PageController onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new PageController(inflater.inflate(R.layout.editor, parent, false)));
  }

  @Override
  public void onBindViewHolder(PageController holder, int position) {
    holder.setText(buffers.get(position));
  }

  @Override
  public int getItemCount() {
    return(PAGE_COUNT);
  }
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

In this case, our model data is an `ArrayList` of `String` objects, representing the text that the user enters into each page's `EditText`. `PAGE_COUNT` caps the number of editors (and pages) at 10, and so we initialize 10 buffers in the `PageAdapter` constructor.

The layout used for the pages — inflated by `onCreateViewHolder()` – is just a full-page multi-line `EditText` widget:

```
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/editor"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:inputType="textMultiLine"
  android:gravity="left|top"
  />
```

(from RecyclerViewPager/PlainRVP/app/src/main/res/layout/editor.xml)

`PageController` is a fairly basic `RecyclerView.ViewHolder`, wrapping our `EditText` and offering a getter and setter for manipulating the text in the editor:

```
package com.commonsware.android.rvp;

import android.support.v7.widget.RecyclerView;
import android.view.View;
import android.widget.EditText;

class PageController extends RecyclerView.ViewHolder {
  private final EditText editor;

  PageController(View itemView) {
    super(itemView);
```

```
    editor=(EditText)itemView.findViewById(R.id.editor);
  }

  void setText(String text) {
    editor.setText(text);
    editor.setHint(editor.getContext().getString(R.string.hint,
      getAdapterPosition()+1));
  }

  String getText() {
    return(editor.getText().toString());
  }
}
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/PageController.java)

In case the buffer is empty (as it is at the outset), we also set the hint of the `EditText` to be the current page's index, adding one to adjust the range to start at 1 rather than 0. The hint text itself is in a string resource, with a `%d` placeholder for the page number:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">RVP Demo</string>
  <string name="hint">Editor #%d</string>

</resources>
```

(from RecyclerViewPager/PlainRVP/app/src/main/res/values/strings.xml)

We use the N-parameter `getString()` method to not only retrieve the `hint` string resource but run it through `String.format()` to populate the placeholders, in this case using `getAdapterPosition()` to determine our page number.

Dealing with Recycling

`RecyclerView` wants to recycle its items. That is in contrast to how the stock `PagerAdapter` implementation work:

- `FragmentPagerAdapter` holds onto every fragment created due to the user's navigation
- `FragmentStatePagerAdapter` holds onto some fragments, but lets others get garbage-collected, to minimize memory consumption

If our pages were read-only, we would not have to worry about recycling. This is how many `RecyclerView` implementations work — they just focus on binding the right data into the right `RecyclerView.ViewHolder` at the right time, based on calls to `onBindViewHolder` in the `RecyclerView.Adapter`.

However, when the `RecyclerView` items are interactive, we need to make sure that we hold onto the changed data, rather than having it be overwritten when we bind fresh data into the recycled item's views.

In `PageAdapter`, we handle this by overriding `onViewDetachedFromWindow()`, which is called when the views of a `PageController` are no longer part of our activity's window. Typically, this will occur as part of scrolling. In our case, we use this opportunity to grab the current contents of that `EditText` widget and update our `buffers` data model to match:

```
@Override
public void onViewDetachedFromWindow(PageController holder) {
  super.onViewDetachedFromWindow(holder);

  buffers.set(holder.getAdapterPosition(), holder.getText());
}
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

Alternatively, you could aim to deal with this more in "real time", such as by using a `TextWatcher` to update the model as the user types. That adds a fair bit of overhead, though.

Dealing with Configuration Changes

We need to make sure that we do not lose what the user types into the pages when we undergo a configuration change. Since our model is a simple `ArrayList` of `String` objects, we can use the saved instance state `Bundle` to hold onto the in-flight information.

A `RecyclerView.Adapter` does not have its own `onSaveInstanceState()` method, but we can add one, then call it from `MainActivity`:

```
@Override
protected void onSaveInstanceState(Bundle state) {
  super.onSaveInstanceState(state);

  Bundle adapterState=new Bundle();
```

```
    adapter.onSaveInstanceState(adapterState);
    state.putBundle(STATE_ADAPTER, adapterState);
  }
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

Here, `MainActivity` provides a fresh `Bundle` to the adapter. This way, values that the adapter wishes to save in the instance state will not collide with anything else the activity would want to save in the instance state, due to accidental key collisions. In this case, this may well be superfluous, but it is a worthwhile practice.

The challenge in our `PageAdapter` is that `buffers` only has text from those `PageController` objects that have been recycled. That will not include the currently-visible page or possibly some adjacent pages.

So, we iterate over all pages and call `findViewHolderForAdapterPosition()` on the `RecyclerView` itself. This will return `null` for any positions for which no `PageController` is presently allocated, or the `PageController` for the position for those positions that are actively being used. For those latter ones, we update the `buffers` to reflect whatever is in the `EditText` widgets, saving that into the instance state `Bundle`:

```
  void onSaveInstanceState(Bundle state) {
    for (int i=0;i<PAGE_COUNT;i++) {
      PageController holder=
        (PageController)pager.findViewHolderForAdapterPosition(i);

      if (holder!=null) {
        buffers.set(i, holder.getText());
      }
    }

    state.putStringArrayList(STATE_BUFFERS, buffers);
  }
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

`MainActivity` has a corresponding `onRestoreInstanceState()` method:

```
  @Override
  protected void onRestoreInstanceState(Bundle state) {
    super.onRestoreInstanceState(state);
```

```
    adapter.onRestoreInstanceState(state.getBundle(STATE_ADAPTER));
  }
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

That delegates the work to the `onRestoreInstanceState()` method on the `PageAdapter`:

```
  void onRestoreInstanceState(Bundle state) {
    buffers=state.getStringArrayList(STATE_BUFFERS);
  }
```

(from RecyclerViewPager/PlainRVP/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

This sets up our `buffers` for use in populating pages again.

Using SnapHelper

`RecyclerViewPager` was first released in 2014. Since then, `RecyclerView` and its supporting classes have evolved. Now, you can get much of the functionality of `RecyclerViewPager` with an ordinary `RecyclerView`, with the assistance of `SnapHelper`. As Lisa Wray profiled in a droidcon NYC 2016 presentation, `SnapHelper` is a utility class that forces swipe gestures to "snap" to certain locations or boundaries. And, there is a `PagerSnapHelper` that, in conjunction with properly-configured `RecyclerView` and items, gives you `ViewPager`-like behavior.

The RecyclerViewPager/PlainSnap sample project is a clone of the `PlainRVP` sample, except that the `RecyclerViewPager` is replaced by a `RecyclerView` and a `PagerSnapHelper`.

There are three requirements of `PagerSnapHelper`. Two are tied to the layouts: both the `RecyclerView` and its items need to have `match_parent` for `android:layout_width` and `android:layout_height`. That was how `PlainRVP` was set up already, though `PlainSnap` swaps in a `RecyclerView` for the `RecyclerViewPager` in `main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView android:id="@+id/pager"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_margin="@dimen/pager_padding"
  android:clipToPadding="false" />
```

(from RecyclerViewPager/PlainSnap/app/src/main/res/layout/main.xml)

The other requirement is that we create an instance of `PagerSnapHelper` and call `attachToRecyclerView()` on it, supplying our `RecyclerView`. This is handled in an updated `MainActivity`:

```
package com.commonsware.android.rvp;

import android.app.Activity;
import android.os.Bundle;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.PagerSnapHelper;
import android.support.v7.widget.RecyclerView;
import android.support.v7.widget.SnapHelper;

public class MainActivity extends Activity {
  private static final String STATE_ADAPTER="adapter";
  private final SnapHelper snapperCarr=new PagerSnapHelper();
  private PageAdapter adapter;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    RecyclerView pager=(RecyclerView)findViewById(R.id.pager);

    pager.setLayoutManager(new LinearLayoutManager(this,
      LinearLayoutManager.HORIZONTAL, false));
    snapperCarr.attachToRecyclerView(pager);

    adapter=new PageAdapter(pager, getLayoutInflater());
    pager.setAdapter(adapter);
  }

  @Override
  protected void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    Bundle adapterState=new Bundle();

    adapter.onSaveInstanceState(adapterState);
    state.putBundle(STATE_ADAPTER, adapterState);
  }

  @Override
  protected void onRestoreInstanceState(Bundle state) {
```

```
    super.onRestoreInstanceState(state);

    adapter.onRestoreInstanceState(state.getBundle(STATE_ADAPTER));
  }
}
```

(from RecyclerViewPager/PlainSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

We hold onto the `PagerSnapHelper` in a field, to ensure that it will not be garbage-collected unexpectedly. *Probably* the `PagerSnapHelper` has sufficient connections to the `RecyclerView` to ensure that it will stay around as long as its associated `RecyclerView` does, but that is not apparent from the API or the documentation.

Beyond that, we configure the `RecyclerView` much as we had configured the `RecyclerViewPager`, and our `PageAdapter` and `PageController` are largely unaffected by the UI switch. In the end, we wind up once again with page-at-a-time horizontal swiping, though this time we can skip the third-party library.

Adding Tabs

Many times, with a pager-style interface, we want an indicator to help the user understand where they are within the range of pages offered by the pager. One of the more popular indicator styles is tabs, as those also provide an alternative navigation option, with the user tapping on tabs to switch to particular pages.

For adding tabs to a `RecyclerView`-powered pager, you need a tab implementation that is not tied inextricably to `ViewPager`, the way `PagerTabStrip` is. At the same time, you need one that is not tied inextricably to some other particular UI setup, the way that `FragmentTabHost` is. Instead, you need tabs that "stick to their knitting" and focus solely on handling the tab UI, giving you the hooks necessary to update your UI based on tab changes, and to update the tabs based on other UI changes.

`TabLayout`, from the Design Support library, is one such tab implementation. While it offers hooks into `ViewPager`, those are optional. You have two main options for using `TabLayout`:

1. Literally use the version from the Design Support library, which will require you to use `appcompat-v7`. This works back to API Level 7, as does `RecyclerView` itself.
2. Use the `TabLayout` from the CWAC-CrossPort library, which is the official `TabLayout` code, with all references to `appcompat-v7` replaced by references

to `Theme.Material` and related native items. However, this limits this cross-ported `TabLayout` to API Level 21 and higher (Android 5.0).

The RecyclerViewPager/TabSnap sample project is a clone of the `PlainSnap` sample, with tabs added, via the `TabLayout` from CWAC-CrossPort. As a result, we need the CWAC-CrossPort dependency and need to raise our `minSdkVersion` to 21:

```
apply plugin: 'com.android.application'

repositories {
  maven {
    url "https://s3.amazonaws.com/repo.commonsware.com"
  }
}

dependencies {
    implementation 'com.android.support:recyclerview-v7:25.1.0'
    implementation 'com.commonsware.cwac:crossport:0.0.2'
}

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 25
        applicationId 'com.commonsware.cwac.rvp.tabsnap'
    }
}
```

(from RecyclerViewPager/TabSnap/app/build.gradle)

The tabs themselves can then go above the `RecyclerView`, in a `vertical` `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:orientation="vertical">

  <com.commonsware.cwac.crossport.design.widget.TabLayout
    android:id="@+id/tabs"
    android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    app:tabMode="scrollable"/>

  <android.support.v7.widget.RecyclerView
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="@dimen/pager_padding"
    android:clipToPadding="false" />
</LinearLayout>
```

(from RecyclerViewPager/TabSnap/app/src/main/res/layout/main.xml)

Next, we need to set up the tab contents in `onCreate()` of `MainActivity`. To do that, we get our hands on the `TabLayout` using `findViewById()`, then iterate through the items in the `PageAdapter` to set up tabs for each:

```
    final TabLayout tabs=(TabLayout)findViewById(R.id.tabs);

    for (int i=0;i<adapter.getItemCount();i++) {
      tabs.addTab(tabs.newTab().setText(adapter.getTabText(this, i)));
    }
```

(from RecyclerViewPager/TabSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

We ask the `PageAdapter` for the text to show in the tab, via a `getTabText()` method:

```
  String getTabText(Context ctxt, int position) {
    return(PageController.getTitle(ctxt, position));
  }
```

(from RecyclerViewPager/TabSnap/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

That, in turn, delegates to a `static` version of the `getTitle()` method on `PageController`, to fill in the string resource template with the proper page number:

```
  static String getTitle(Context ctxt, int position) {
    return(ctxt.getString(R.string.hint, position+1));
  }
```

(from RecyclerViewPager/TabSnap/app/src/main/java/com/commonsware/android/rvp/PageController.java)

We now need to add code in `onCreate()` of `MainActivity` to tie the navigation together:

- When the user taps a tab, we need to update the pager

- When the user swipes the pager, we need to update the tabs to show the new selection

This is handled by event listeners:

```
tabs.addOnTabSelectedListener(new TabLayout.OnTabSelectedListener() {
  @Override
  public void onTabSelected(TabLayout.Tab tab) {
    pager.smoothScrollToPosition(tab.getPosition());
  }

  @Override
  public void onTabUnselected(TabLayout.Tab tab) {
    // unused
  }

  @Override
  public void onTabReselected(TabLayout.Tab tab) {
    // unused
  }
});

pager.setOnScrollListener(new RecyclerView.OnScrollListener() {
  @Override
  public void onScrolled(RecyclerView recyclerView, int dx, int dy) {
    int tab=layoutManager.findFirstCompletelyVisibleItemPosition();

    if (tab>=0 && tab<tabs.getTabCount()) {
      tabs.getTabAt(tab).select();
    }
  }
});
```

(from RecyclerViewPager/TabSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

When a tab is selected, our anonymous `TabLayout.OnTabSelectedListener` implementation will get control in `onTabSelected()`. There, we tell the `RecyclerView` to scroll to show a particular position, tied to the position of the selected tab.

Similarly, when the user scrolls the pager, we need to update the tabs to show the new selection. To do that, we take advantage of the `findFirstCompletelyVisibleItemPosition()` method on `LinearLayoutManager`. As the (lengthy) method name suggests, this returns the position of the first item that is completely visible within the pager. This might return –1, if we are in the middle of

a swipe, as no item may be *completely* visible at that point. But, once we get a plausible value, we tell the `TabLayout` to select that tab.

`RecyclerViewPager` has a more sophisticated algorithm for integrating with the official Design Support library implementation of `TabLayout`. `RecyclerViewPager` calculates the position of the tab highlight, based upon the current swipe position, and updates that. This provides visual feedback within the tabs while the swipe is going on. The approach shown in the sample app has the effect of only updating the tabs once the swipe is completed.

Changing the Mix of Pages

A major problem with `ViewPager` — or, more accurately, the stock `PagerAdapter` implementations — was when we wanted to add, move, or remove pages on the fly, such as by the user tapping some sort of "add tab" action bar item. To pull this off required a custom `PagerAdapter`, as neither `FragmentPagerAdapter` nor `FragmentStatePagerAdapter` were well-suited for this scenario.

`RecyclerView` can readily handle such changes in its roster of items, and so we can add, move, and remove pages easily enough. However, it does require a somewhat more elaborate example, not so much due to `RecyclerView`, but for the rest of our business logic. For example, all the prior samples hard-coded `getItemCount()` to return 10 (by way of a `PAGE_COUNT` value), and that will no longer be the case as we add and remove pages.

The `RecyclerViewPager/DynSnap` sample project is a clone of the `TabSnap` sample, with four action bar items, to add, "split", swap, and remove pages.

Using a Richer Model

In our previous samples, the tab title was tied to the tab position. That worked well, since our tabs all had fixed positions. Now, though, we want to add, move, and remove tabs, which means the positions of the existing tabs will change. We do not want to change the tab titles just because the positions change, as that will be very confusing for the user. So, our data model is more than just the text that the user types in — now it needs to include the title associated with that text.

To that end, `DynSnap` sets up an actual model object, called `EditBuffer`, to track both of those values:

```
package com.commonsware.android.rvp;

import android.os.Parcel;
import android.os.Parcelable;

class EditBuffer implements Parcelable {
  private String prose;
  final private String title;

  EditBuffer(String title) {
    this(title, "");
  }

  EditBuffer(String title, String prose) {
    this.prose=prose;
    this.title=title;
  }

  protected EditBuffer(Parcel in) {
    prose=in.readString();
    title=in.readString();
  }

  @Override
  public String toString() {
    return(title);
  }

  String getProse() {
    return(prose);
  }

  void setProse(String prose) {
    this.prose=prose;
  }

  @Override
  public int describeContents() {
    return 0;
  }

  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(prose);
    dest.writeString(title);
  }

  @SuppressWarnings("unused")
```

```
  public static final Parcelable.Creator<EditBuffer> CREATOR=
    new Parcelable.Creator<EditBuffer>() {
    @Override
    public EditBuffer createFromParcel(Parcel in) {
      return(new EditBuffer(in));
    }

    @Override
    public EditBuffer[] newArray(int size) {
      return(new EditBuffer[size]);
    }
  };
}
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/EditBuffer.java)

`EditBuffer` is `Parcelable`, so we will be able to hold onto this information in the saved instance state `Bundle`, for dealing with configuration changes.

The `buffers` field in `PageAdapter` now is an `ArrayList` of `EditBuffer` objects, and our original constructor and other methods are updated to match that:

```
PageAdapter(RecyclerView pager, LayoutInflater inflater) {
  this.pager=pager;
  this.inflater=inflater;

  for (int i=0;i<10;i++) {
    buffers.add(new EditBuffer(getNextTitle()));
  }
}

@Override
public PageController onCreateViewHolder(ViewGroup parent, int viewType) {
  return(new PageController(inflater.inflate(R.layout.editor, parent, false)));
}

@Override
public void onBindViewHolder(PageController holder, int position) {
  holder.setText(buffers.get(position).getProse());
  holder.setTitle(buffers.get(position).toString());
}

@Override
public int getItemCount() {
  return(buffers.size());
}

String getTabText(int position) {
  return(buffers.get(position).toString());
}

@Override
public void onViewDetachedFromWindow(PageController holder) {
```

```
    super.onViewDetachedFromWindow(holder);

    if (holder.getAdapterPosition()>=0) {
      buffers.get(holder.getAdapterPosition()).setProse(holder.getText());
    }
  }

  private String getNextTitle() {
    return(pager.getContext().getString(R.string.hint, nextTabNumber++));
  }

  void onSaveInstanceState(Bundle state) {
    for (int i=0;i<getItemCount();i++) {
      updateProse(i);
    }

    state.putParcelableArrayList(STATE_BUFFERS, buffers);
  }

  void onRestoreInstanceState(Bundle state) {
    buffers=state.getParcelableArrayList(STATE_BUFFERS);
  }
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

We pull out the "update-the-`EditBuffer`-based-on-the-holder-contents" logic into a separate `updateProse()` method, that we will reuse elsewhere:

```
  private void updateProse(int position) {
    PageController holder=
      (PageController)pager.findViewHolderForAdapterPosition(position);

    if (holder!=null) {
      buffers.get(position).setProse(holder.getText());
    }
  }
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

Also note that when we set up the buffers in the constructor, we call a `getNextTitle()` method. This tracks the number of titles that we have generated, via a `nextTabNumber` field, and creates a new title with a new number on each call. This further decouples our tab titles from any concept of tab positions.

Adding Action Bar Items

We have four vector assets in `res/drawable/` for our four action bar items:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/add"
```

```
    android:icon="@drawable/ic_add_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@string/menu_add" />
  <item
    android:id="@+id/split"
    android:icon="@drawable/ic_split_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@string/menu_split" />
  <item
    android:id="@+id/swap"
    android:icon="@drawable/ic_swap_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@string/menu_swap" />
  <item
    android:id="@+id/remove"
    android:icon="@drawable/ic_remove_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@string/menu_remove" />
</menu>
```

(from RecyclerViewPager/DynSnap/app/src/main/res/menu/actions.xml)

Those items get inflated and applied in `MainActivity`:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);
    remove=menu.findItem(R.id.remove);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
      case R.id.add:
        add();
        return(true);

      case R.id.split:
        split();
        return(true);

      case R.id.swap:
        swap();
        return(true);

      case R.id.remove:
```

```
      remove();
      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

Each of the items gets tied to its own dedicated method (e.g., `R.id.add` triggers a call to `add()`).

Note that we find and hold onto the `remove` `MenuItem`, caching its value in a field. We are going to need to disable this `MenuItem` when we can no longer allow the user to remove pages. In the case of this app, we prevent the user from removing all the pages, so once we are down to our last page, we need to disable the `MenuItem`, re-enabling it when the user adds new pages.

Adding Pages

The "add" and "split" operations both do the same thing: add a page. One difference is where they add it. "Add" adds a page before the current one. "Split" adds a page after the current one.

The `add()` and `split()` methods therefore do very similar things:

- Determine where the new tab's position will be
- Tell the adapter to do its work for managing the data model and pager
- Adds a new tab for the new position
- Updates the `remove` `MenuItem`, in case it had been disabled and should now be enabled

```
private void add() {
  int current=getCurrentPosition();

  adapter.insert(current);
  tabs.addTab(tabs.newTab().setText(adapter.getTabText(current)), current);
  updateRemoveMenuItem();
}

private void split() {
  int newPosition=getCurrentPosition()+1;

  adapter.clone(newPosition-1);
  tabs.addTab(tabs.newTab().setText(adapter.getTabText(newPosition)), newPosition);
  updateRemoveMenuItem();
}
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

Here, `updateRemoveMenuItem()` simply sets the enabled state of the `remove` item based on our page count:

```
private void updateRemoveMenuItem() {
  remove.setEnabled(adapter.getItemCount()>1);
}
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

Another difference between "add" and "split" is what the new page has for content. "Add" adds an empty page. "Split" clones the prose from the selected page and puts that prose in the new page as well.

That is why `add()` on `MainActivity` calls an `insert()` method on `PageAdapter` and `split()` calls `clone()`, because the `PageAdapter` has different behavior for each:

```
void insert(int position) {
  buffers.add(position, new EditBuffer(getNextTitle()));
  notifyItemInserted(position);
}

void clone(int position) {
  updateProse(position);

  EditBuffer newBuffer=new EditBuffer(getNextTitle(),
    buffers.get(position).getProse());

  buffers.add(position+1, newBuffer);
  notifyItemInserted(position+1);
}
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

In both cases, a new `EditBuffer` gets added to the `buffers`, and we call `notifyItemInserted()` to let the `RecyclerView` know of the data model change. In the case of `clone()`, we also update the `EditBuffer` for the page to be cloned, then use that data to populate the new `EditBuffer`.

Moving Pages

The `swap()` method on `MainActivity` flips the positions of two pages. Normally, this will be the active tab and the next tab, but if the active tab is the *last* tab, the pair will be the active tab and the previous tab.

Hence, `swap()` identifies the right pair of positions, tells the `PageAdapter` to swap the contents, then swaps the titles of the two affected tabs:

```
private void swap() {
  int first=getCurrentPosition();
  int second;

  if (first>=adapter.getItemCount()-1) {
    second=first;
    first--;
  }
  else {
    second=first+1;
  }

  adapter.swap(first, second);

  TabLayout.Tab firstTab=tabs.getTabAt(first);
  TabLayout.Tab secondTab=tabs.getTabAt(second);
  CharSequence firstText=firstTab.getText();

  firstTab.setText(secondTab.getText());
  secondTab.setText(firstText);
}
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

The `swap()` method on `PageAdapter` needs to do three things:

- Update the `buffers` to reflect the swapped pages
- Update the existing `PageController` objects for those pages, if they exist
- Tell the `RecyclerView` that the contents of these two positions changed, to ensure that they get redrawn

```
void swap(int first, int second) {
  EditBuffer firstBuffer=buffers.get(first);
  EditBuffer secondBuffer=buffers.get(second);

  buffers.set(first, secondBuffer);
  buffers.set(second, firstBuffer);

  PageController holder=
    (PageController)pager.findViewHolderForAdapterPosition(first);

  if (holder!=null) {
    holder.setText(secondBuffer.getProse());
```

```
    }

    holder=(PageController)pager.findViewHolderForAdapterPosition(second);

    if (holder!=null) {
      holder.setText(firstBuffer.getProse());
    }

    notifyItemChanged(first);
    notifyItemChanged(second);
  }
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

Removing Pages

Removing a page involves:

- Removing the `EditBuffer` from the data model
- Removing the associated item in the `RecyclerView`
- Removing the associated tab
- Updating the `remove` `MenuItem`, if we are down to our last page

The latter two tasks are handled by `remove()` on `MainActivity`:

```
  private void remove() {
    final int current=getCurrentPosition();

    tabs.removeTabAt(current);
    adapter.remove(current);

    if (current<adapter.getItemCount()) {
      pager.scrollToPosition(current);
    }

    updateRemoveMenuItem();
  }
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/MainActivity.java)

The former two tasks are handled by `remove()` on `PageAdapter`:

```
  void remove(int position) {
    buffers.remove(position);
    notifyItemRemoved(position);
  }
```

(from RecyclerViewPager/DynSnap/app/src/main/java/com/commonsware/android/rvp/PageAdapter.java)

Declaring a LayoutManager in the Layout

The typical way of setting up a `LayoutManager` is to call `setLayoutManager()` on the `RecyclerView`. There may be no choice in some cases, if the `LayoutManager` that you wish to use has a custom constructor that you need.

However, in some cases, where the `LayoutManager` has adequate setters to configure it beyond the constructor, you can set up the `LayoutManager` in your layout resource file. Just add the `app:layoutManager` attribute to the `RecyclerView` element, with the value being:

- the bare class name of a standard `LayoutManager`, such as `LinearLayoutManager`, or
- the fully-qualified class name of some custom `LayoutManager`, one that you wrote or are using from a library

The `app` XML namespace (`xmlns:app="http://schemas.android.com/apk/res-auto"`) can be added to the layout resource file by an Android Studio quick-fix.

Transcript Mode

"Transcript mode" with `ListView` means that the `ListView` automatically scrolls to keep the bottom entries in view. This is useful for things like chat transcripts, where you want to append new messages and show those new messages to the user, and the new messages (traditionally) go below older messages. This approach — using `setTranscriptMode()` — has been used a couple of times in this book, such as in the chapter profiling various event bus implementations.

`RecyclerView` also supports this approach, though it is set up somewhat differently.

The `RecyclerView/Transcript` sample project is a clone of the `EventBus/GreenRobot3` sample project, where the `ListView` is replaced by a `RecyclerView`.

In a layout resource, we declare our `RecyclerView`, with a few interesting attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView android:id="@+id/transcript"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent"
app:layoutManager="LinearLayoutManager"
app:reverseLayout="true"
app:stackFromEnd="true" />
```

(from RecyclerView/Transcript/app/src/main/res/layout/main.xml)

Specifically, we have:

- `app:layoutManager` defines the layout manager that we want to use, in this case a `LinearLayoutManager`
- `app:reverseLayout="true"` and `app:stackFromEnd="true"`, which combine to give us a transcript-style effect

Note that those latter two attributes are actually documented on `LinearLayoutManager`, not on `RecyclerView` (and, even then, they are largely undocumented). Those attributes may not work with other layout managers (e.g., `GridLayoutManager`).

Beyond that, the rest of the code works pretty much as the original sample app did, except that we need to substitute in `RecyclerView` equivalents for the `ListView` functionality. So, for example, rather than inheriting from `ListViewFragment`, the `EventLogFragment` is now a regular `Fragment`, where we inflate our layout in `onCreateView()` and attach its `EventLogAdapter` in `onViewCreated()`:

```
@Nullable
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
  return(inflater.inflate(R.layout.main, container, false));
}

@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  if (adapter==null) {
    adapter=new EventLogAdapter();
  }

  RecyclerView transcript=(RecyclerView)view.findViewById(R.id.transcript);

  transcript.setAdapter(adapter);
}
```

(from RecyclerView/Transcript/app/src/main/java/com/commonsware/android/rv/transcript/EventLogFragment.java)

The old `ArrayAdapter` is gone, so `EventLogAdapter` is now a `RecyclerView.Adapter`, managing a series of `RowHolder` objects for our rows:

```
class EventLogAdapter extends RecyclerView.Adapter<RowHolder> {
  private final ArrayList<RandomEvent> events=new ArrayList<>();

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent,
                                      int viewType) {
    View row=
      getActivity()
        .getLayoutInflater()
        .inflate(android.R.layout.simple_list_item_1, parent, false);

    return(new RowHolder(row));
  }

  @Override
  public void onBindViewHolder(RowHolder holder,
                               int position) {
    holder.bind(events.get(position));
  }

  @Override
  public int getItemCount() {
    return(events.size());
  }

  void add(RandomEvent event) {
    events.add(event);
    notifyItemInserted(getItemCount());
  }
}

static class RowHolder extends RecyclerView.ViewHolder {
  private static final DateFormat fmt=
    new SimpleDateFormat("HH:mm:ss", Locale.US);
  private final TextView tv;

  RowHolder(View itemView) {
    super(itemView);

    tv=(TextView)itemView.findViewById(android.R.id.text1);
  }
```

```
    void bind(RandomEvent event) {
      tv.setText(String.format("%s = %x", fmt.format(event.when),
        event.value));
    }
  }
```

(from RecyclerView/Transcript/app/src/main/java/com/commonsware/android/rv/transcript/EventLogFragment.java)

And, when we get a `RandomEvent`, we `add()` that to our `EventLogAdapter`, which adds it to its array of events and updates the `RecyclerView` to match:

```
@Subscribe(threadMode = ThreadMode.MAIN)
public void onRandomEvent(final RandomEvent event) {
  adapter.add(event);
}
```

(from RecyclerView/Transcript/app/src/main/java/com/commonsware/android/rv/transcript/EventLogFragment.java)

Advanced Uses of WebView

Android uses the WebKit browser engine as the foundation for both its Browser application and the `WebView` embeddable browsing widget. The Browser application, of course, is something Android users can interact with directly; the `WebView` widget is something you can integrate into your own applications for places where an HTML interface might be useful.

Earlier in this book, we saw a simple integration of a `WebView` into an Android activity, with the activity dictating what the browsing widget displayed and how it responded to links.

Here, we will expand on this theme, and show how to more tightly integrate the Java environment of an Android application with the JavaScript environment of WebKit.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one covering `WebView`. Some of the samples use `LocationManager` for obtaining a GPS fix.

Friends with Benefits

When you integrate a `WebView` into your activity, you can control what Web pages are displayed, whether they are from a local provider or come from over the Internet, what should happen when a link is clicked, and so forth. And between `WebView`, `WebViewClient`, and `WebSettings`, you can control a fair bit about how the embedded browser behaves. Yet, by default, the browser itself is just a browser,

capable of showing Web pages and interacting with Web sites, but otherwise gaining nothing from being hosted by an Android application.

However, `WebView` offers a few options for more tightly integrating the Java and JavaScript realms, so Web content can call into your app to get data to display, and your app can push data into the Web page for JavaScript to render.

Unfortunately, the techniques for doing this have changed over the years. Partially that is due to changes in `WebView`, particularly starting with Android 4.4. But, some of the changes are due to security issues, particularly when you are loading arbitrary content, such as Web-based ads from an ad network, into your `WebView`.

The following sections will go over four separate sample apps. All do the same thing: provide data about the ambient light level, using the sensor on the phone or tablet, to the Web page for rendering. The differences are whether we are pushing data from Java into JavaScript (e.g., as the light level changes), or whether we are pulling data from Java using JavaScript (e.g., in response to a user tapping on something in the Web page). Also, we will see two approaches to push and two approaches to pull.

JavaScript Calling Java: addJavascriptInterface()

The `addJavascriptInterface()` method on `WebView` allows you to inject a Java object into the `WebView`, exposing its methods, so they can be called by JavaScript loaded by the Web content in the `WebView` itself.

Now you have the power to provide access to a wide range of Android features and capabilities to your `WebView`-hosted content. If you can access it from your activity, and if you can wrap it in something convenient for use by JavaScript, your Web pages can access it as well.

The WebKit/SensorPull sample project demonstrates using `addJavascriptInterface()` to pull light sensor data into a Web page to display to the user.

For all four of these sample apps, the UI is just a `WebView`:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView android:id="@+id/webkit"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
```

```
</WebView>
```

(from WebKit/SensorPull/app/src/main/res/layout/main.xml)

In `onCreate()`, we set things up:

```
@SuppressLint({"AddJavascriptInterface", "SetJavaScriptEnabled"})
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
  light=mgr.getDefaultSensor(Sensor.TYPE_LIGHT);

  wv=(WebView)findViewById(R.id.webkit);
  wv.getSettings().setJavaScriptEnabled(true);
  wv.addJavascriptInterface(jsInterface, "LIGHT_SENSOR");
  wv.loadUrl("file:///android_asset/index.html");
}
```

(from WebKit/SensorPull/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

Specifically, we:

- Get a `SensorManager` and the appropriate `Sensor` for measuring ambient light levels, saving those in fields
- Get our `WebView` via `findViewById()`
- Enable JavaScript (which comes disabled by default), using `getSettings()` and `setJavaScriptEnabled(true)`
- Call `addJavascriptInterface()`, and
- Load a Web page from assets (which we will examine shortly)

Because we are enabling JavaScript, Lint will complain that this poses security risks, so `onCreate()` has a `@SuppressLint` annotation for `SetJavaScriptEnabled` to indicate that we are aware of the risks. Similarly, because we are calling `addJavascriptInterface()`, Lint will complain that this poses even more security risks. So, `@SuppressLint` suppresses both the `SetJavaScriptEnabled` warning and the `AddJavascriptInterface` warning.

Also, you may notice that there is a significant debate within the Android SDK as to whether the "s" in "JavaScript" gets capitalized or not. In general, it does, but `addJavascriptInterface()` shipped in API Level 1 with a lowercase "s" in its name,

and so that method, and variations of it (e.g., the `AddJavascriptInterface` annotation) will use a lowercase "s". Eventually, you just get used to this.

`addJavascriptInterface()` takes two parameters: a Java object to inject into the JavaScript of the Web page, and a `String` that is the name by which JavaScript can reference that object. So, we have a `jsInterface` object that JavaScript can reference via `LIGHT_SENSOR`.

`jsInterface` is an instance of `JSInterface`, a `static` nested class inside `MainActivity`:

```
private static class JSInterface {
  float lux=0.0f;

  private void updateLux(float lux) {
    this.lux=lux;
  }

  @JavascriptInterface
  public String getLux() {
    return(String.format(Locale.US, "{\"lux\": %f}", lux));
  }
}
```

(from WebKit/SensorPull/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

It just has a getter and setter around the current light level, which is a `float` named `lux` (referring to the unit of brightness used for the values coming from the ambient light sensor). The getter, however, has two interesting traits:

- It returns a `String`, representing a JSON object wrapped around the `lux` value
- It is annotated with `@JavascriptInterface`

This annotation is required of apps with `android:targetSdkVersion` set to 17 or higher, though it is a good idea to start using it anyway. With such an `android:targetSdkVersion`, in an app running on an Android 4.2 or higher device, *only* `public` methods with the `@JavascriptInterface` annotation will be accessible by JavaScript code. On earlier devices, or with an earlier `android:targetSdkVersion`, *all* `public` methods on the `JsInterface` object would be accessible by JavaScript, including those inherited from superclasses like `Object`. Note that your build target (i.e., `compileSdkVersion` in Android Studio) will need to be Android 4.2 or higher in order to reference the `@JavascriptInterface` annotation.

The reason for returning a JSON object (in string form), rather than just the `float`, is for two reasons:

1. For more complex APIs, you cannot pass into JavaScript an arbitrary Java object. All return types from `@JavascriptInterface` objects need to be something that JavaScript can use, and a simple way to do that is to create data structures in JSON.
2. `float` did not seem to work well as a return type, as it always seemed to turn into `0.0` on the JavaScript side, for unknown reasons

We register with the `SensorManager` to find out when the light level changes, via `registerListener()` (in `onStart()`) and `unregisterListener()` (in `onStop()`):

```
@Override
protected void onStart() {
  super.onStart();

  mgr.registerListener(this, light, SensorManager.SENSOR_DELAY_UI);
}

@Override
protected void onStop() {
  mgr.unregisterListener(this);

  super.onStop();
}
```

(from WebKit/SensorPull/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

That, in turn, will trigger a call to `onSensorChanged()` when the light level changes. There, we pass the light level (the first `float` out of the `values` array from the `SensorEvent`) to the `JsInterface` instance, ready to be retrieved by the JavaScript code in our Web page:

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
  jsInterface.updateLux(sensorEvent.values[0]);
}

@Override
public void onAccuracyChanged(Sensor sensor, int i) {
  // unused
}
```

(from WebKit/SensorPull/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

In the Web page, we set it up to show the current light level, starting with a value of `0.0`. When the user taps on the `Light Level` caption, we call a `pull()` JavaScript function, which:

- Calls our `getLux()` method on the `LIGHT_SENSOR` global object
- Parses our JSON into a JavaScript object
- Calls an `update_lux()` function to update the `lux` span with the new light level

```
<html>
<head>
<title>Android Light Sensor Demo</title>
<script language="javascript">
  function update_lux(lux) {
    document.getElementById("lux").innerHTML=lux;
  }

  function pull() {
    var result=JSON.parse(LIGHT_SENSOR.getLux());

    update_lux(result.lux);
  }
</script>
</head>
<body>
<p><b><a onClick="pull()">Light Level</a>: </b><span id="lux">0.0</span> lux</p>
</body>
</html>
```

(from WebKit/SensorPull/app/src/main/assets/index.html)

If you run the app, you get our trivial Web page in the `WebView`:

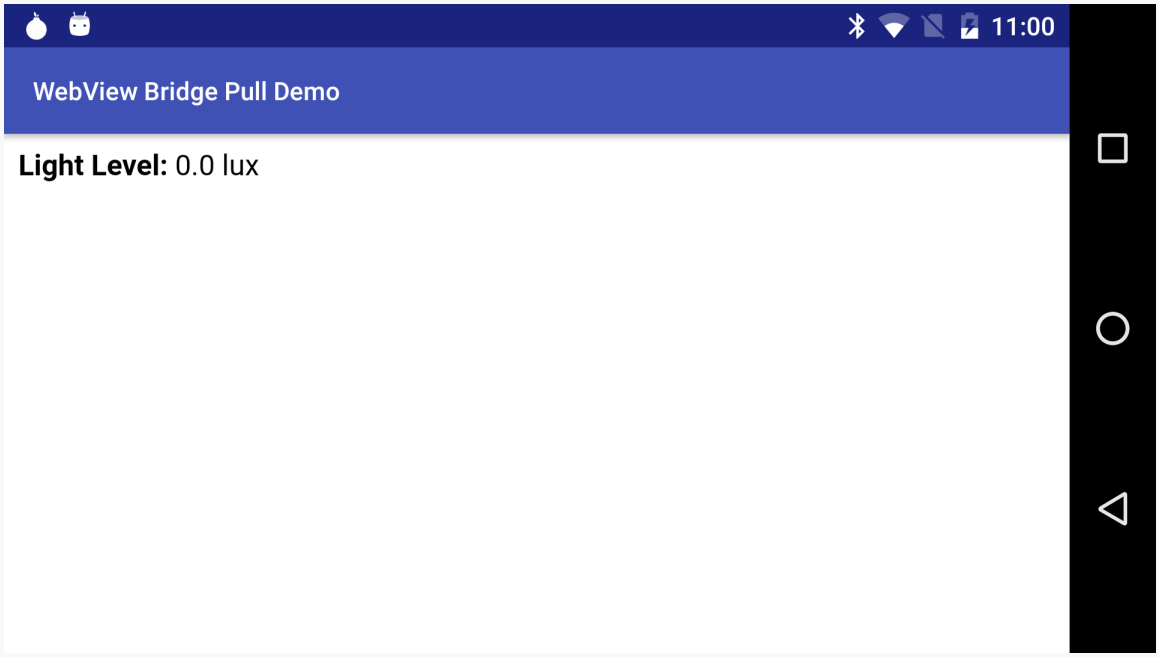


Figure 552: SensorPull Demo, As Initially Launched

Tapping on the words “Light Level” will cause JavaScript to request the light level, updating the page to match:

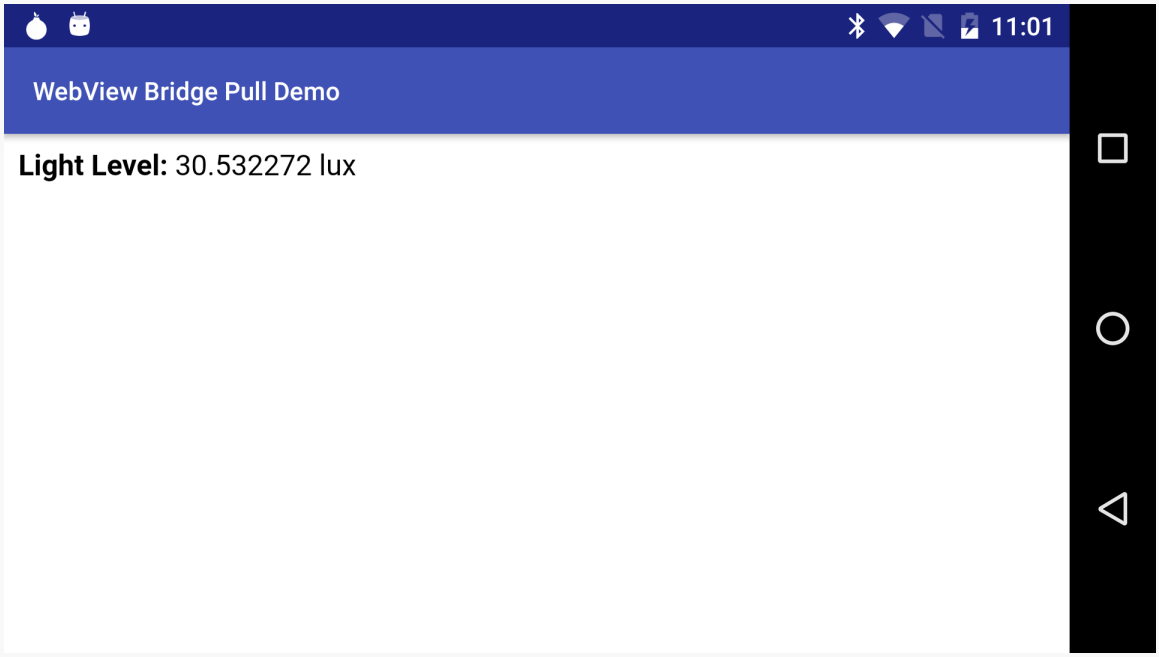


Figure 553: SensorPull Demo, Showing Light Level

Note that this sample app will only work on devices with an ambient light sensor. It is rather likely that the app will crash spectacularly on devices lacking such a sensor.

Unfortunately, `addJavascriptInterface()` opens up a number of security issues, outlined later in this chapter. Where possible, avoid the use of this API.

Java Calling JavaScript: loadUrl() and evaluateJavascript()

Now that we have seen how JavaScript can call into Java, it would be nice if Java could somehow call out to JavaScript. Well, as luck would have it, we can do that too. This is a good thing, otherwise, this would be a really weak section of the book.

What is unusual is how you call out to JavaScript. One might imagine there would be an `evaluateJavaScript()` counterpart to `addJavascriptInterface()`, where you could supply some JavaScript source and have it executed within the context of the currently-loaded Web page.

Actually, there *is* such a method on Android 4.4. However, earlier versions of Android lacked that method. Instead, on older versions of Android, given your snippet of JavaScript source to execute, you call `loadUrl()` on your `WebView`, as if you

were going to load a Web page, but you put `javascript:` in front of your code and use that as the "address" to load.

If you have ever created a "bookmarklet" for a desktop Web browser, you will recognize this technique as being the Android analogue – the `javascript:` prefix tells the browser to treat the rest of the address as JavaScript source, injected into the currently-viewed Web page.

The WebKit/SensorPush sample project expands upon the `SensorPull` app. This time, though, in addition to pulling via `addJavascriptInterface()`, we support pushing light levels as sensor readings come in.

That comes courtesy of a revised `onSensorChanged()` method:

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
  float lux=sensorEvent.values[0];

  jsInterface.updateLux(lux);

  String js=String.format(Locale.US, "update_lux(%f)", lux);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.KITKAT) {
    wv.evaluateJavascript(js, null);
  }
  else {
    wv.loadUrl("javascript:"+js);
  }
}
```

(from WebKit/SensorPush/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

Before, we just updated the `JsInterface` object with the new light level. Now, we also format a JavaScript call to `update_lux()`, supplying our light level. Then, based on Android OS version (`Build.VERSION.SDK_INT`), we either call `evaluateJavascript()` or `loadUrl()`, the latter also employing the `javascript:` scheme.

Because we had pulled out `update_lux()` as a separate function before, our HTML and JavaScript does not need to change at all:

```
<html>
<head>
<title>Android Light Sensor Demo</title>
<script language="javascript">
```

```
  function update_lux(lux) {
    document.getElementById("lux").innerHTML=lux;
  }

  function pull() {
    var result=JSON.parse(LIGHT_SENSOR.getLux());

    update_lux(result.lux);
  }
</script>
</head>
<body>
<p><b><a onClick="pull()">Light Level</a>: </b><span id="lux">0.0</span> lux</p>
</body>
</html>
```

(from WebKit/SensorPush/app/src/main/assets/index.html)

If you run this sample app, you will find that the Web page updates in real time as you wave your hand in front of the light sensor, shine a light on that sensor, etc.

Java Calling JavaScript: WebMessage

Both of those techniques have worked since API Level 1. But, as mentioned, `addJavascriptInterface()` has security issues. Also, `evaluateJavascript()` (or its `loadUrl()` equivalent) requires the Java code to know what functions are available in the Web page. That may tie the Java and JavaScript more tightly than you might like.

Android 6.0 introduced another pair of options for communicating between Java and JavaScript — `WebMessage` and `WebMessagePort` — that try to eliminate these issues.

The simpler of the two approaches is `WebMessage`. Instead of calling `evaluateJavascript()` or `loadUrl()`, you create a `WebMessage` object and call `postWebMessage()` to deliver it to the JavaScript in your Web page. So, in the WebKit/SensorMessage sample project, we have an updated `onSensorChanged()` method that does this:

```
  @Override
  public void onSensorChanged(SensorEvent sensorEvent) {
    float lux=sensorEvent.values[0];

    jsInterface.updateLux(lux);

    String js=String.format(Locale.US, "update_lux(%f)", lux);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
```

```
      wv.postWebMessage(new WebMessage(jsInterface.getLux()),
        Uri.EMPTY);
    }
    else if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.KITKAT) {
      wv.evaluateJavascript(js, null);
    }
    else {
      wv.loadUrl("javascript:"+js);
    }
  }
```

(from WebKit/SensorMessage/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

The `WebMessage` constructor that we are using here takes a simple string that is the content of the message. In this case, it is the JSON object wrapping our light level in lux, using the same `getLux()` method that JavaScript can call on our `JsInterface` instance that we registered via `addJavascriptInterface()`.

`postWebMessage()` takes two parameters. The first is the `WebMessage` to deliver to the page. The other is supposed to be the `Uri` of the Web page. This is supposed to be used to confirm that you are sending the message to the page that you think you are sending it to.

Unfortunately, this is not behaving especially well.

It only works as advertised for `http`/`https` URLs, or for data that you load using `loadDataWithBaseURL()` and supply some `http` or `https` URL. If you load from a `file` URL, as we are doing here, you cannot use the actual URL. Instead, you have to use `Uri.EMPTY`, which is a "wildcard" that skips over this test, which is what we use here. Apparently, this is all working as intended.

To receive these messages, our JavaScript needs to define an `onmessage()` global function:

```
<html>
<head>
<title>Android Light Sensor Demo</title>
<script language="javascript">
  function update_lux(lux) {
    document.getElementById("lux").innerHTML=lux;
  }

  function parse(json) {
    var result=JSON.parse(json);

    update_lux(result.lux);
  }
```

```
  function pull() {
    parse(LIGHT_SENSOR.getLux());
  }

  onmessage = function (e) {
    parse(e.data);
  }
</script>
</head>
<body>
<p><b><a onClick="pull()">Light Level</a>: </b><span id="lux">0.0</span> lux</p>
</body>
</html>
```

(from WebKit/SensorMessage/app/src/main/assets/index.html)

This receives the HTML Web message equivalent of the `WebMessage` that we posted. The `data` field on the supplied event object (`e` in the sample) contains our string. So, we turn around and `parse()` it, just as we would `parse()` the JSON we got from calling `getLux()` on our `LIGHT_SENSOR`.

If you run this sample app on an Android 6.0+ device, you should get the same results as with `SensorPush`, where the light level changes automatically. However, in this case, we will be using code that relies upon `WebMessage` and `postWebMessage()`, instead of `evaluateJavascript()` or `loadUrl()`. In particular, our Java code does not need to know anything about the internal workings of the JavaScript (e.g., function names) — it just passes over the message and relies on the JavaScript to have registered itself appropriately to receive the message.

JavaScript Calling Java: WebMessagePort

What would be nice is to use this `WebMessage` system to be able to replace `addJavascriptInterface()` and allow JavaScript to call back into Java. This is possible, but it is fairly complex.

For our Java code to receive messages sent to it from JavaScript, we need to do three things:

1. Call `createWebMessageChannel()` on the `WebView`. This creates a private communications channel between us and our target Web page. It returns a two-element `WebMessagePort` array. Index `0` of that array is our end of the channel; index `1` is the JavaScript end of the channel.

2. Call `setWebMessageCallback()` on our `WebMessagePort`, supplying a `WebMessageCallback` that will be called with `onMessage()` when a message arrives on the port from JavaScript.
3. Send the other `WebMessagePort` to JavaScript using a `WebMessage`.

In the WebKit/SensorPort sample project, this is handled by an `initPort()` method:

```
@TargetApi(Build.VERSION_CODES.M)
private void initPort() {
  final WebMessagePort[] channel=wv.createWebMessageChannel();

  port=channel[0];
  port.setWebMessageCallback(new WebMessagePort.WebMessageCallback() {
    @Override
    public void onMessage(WebMessagePort port, WebMessage message) {
      postLux();
    }
  });

  wv.postWebMessage(new WebMessage("", new WebMessagePort[]{channel[1]}),
        Uri.EMPTY);
}
```

(from WebKit/SensorPort/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

The `WebMessage` constructor that we use this time takes two parameters: an arbitrary string (here, just set to `""`, as we are not using it) and a one-element `WebMessagePort` array containing the JavaScript end of the communications channel.

However, we cannot do any of this work until the Web page is ready to be used. Otherwise, the JavaScript code will not receive our `WebMessage`, since it is not yet ready.

So, we have to postpone calling `initPort()` until a `WebViewClient` is called with `onPageFinished()`, as we do in `onCreate()`:

```
@SuppressLint({"AddJavascriptInterface", "SetJavaScriptEnabled"})
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
  light=mgr.getDefaultSensor(Sensor.TYPE_LIGHT);
```

```
    wv=(WebView)findViewById(R.id.webkit);
    wv.getSettings().setJavaScriptEnabled(true);
    wv.addJavascriptInterface(jsInterface, "LIGHT_SENSOR");

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
      wv.setWebViewClient(new WebViewClient() {
        @Override
        public void onPageFinished(WebView view, String url) {
          initPort();
        }
      });
    }

    wv.loadUrl(URL);
  }
```

(from WebKit/SensorPort/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

Our `WebMessageCallback`, upon receipt of a `WebMessage` from the JavaScript, calls a `postLux()` method. We are just using the existence of the message as a "ping" from the JavaScript to Java, asking for us to send it the ambient light level. So, in `postLux()`, we create a `WebMessage` and send it to JavaScript... but not via the `postWebMessage()` method on `WebView`. Instead, we use *our* end of the `WebMessagePort` communications channel, calling `postMessage()` on it, in a `postLux()` method:

```
  @TargetApi(Build.VERSION_CODES.M)
  private void postLux() {
    port.postMessage(new WebMessage(jsInterface.getLux()));
  }
```

(from WebKit/SensorPort/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

So, when the JavaScript sends a `WebMessage` to Java, Java sends a `WebMessage` right back, supplying the light level JSON.

To prove that this is working, this sample comments out the automated push of the light level in `onSensorChanged()` — ordinarily, we would call `postLux()` to push over the light level when we get it:

```
  @Override
  public void onSensorChanged(SensorEvent sensorEvent) {
    float lux=sensorEvent.values[0];

    jsInterface.updateLux(lux);
```

```
    String js=String.format(Locale.US, "update_lux(%f)", lux);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
      // postLux();
    }
    else if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.KITKAT) {
      wv.evaluateJavascript(js, null);
    }
    else {
      wv.loadUrl("javascript:"+js);
    }
  }
```

(from WebKit/SensorPort/app/src/main/java/com/commonsware/android/webkit/bridge/MainActivity.java)

In JavaScript, our `onmessage` global function is now a bit more complex as well. We get our end of the communications channel by retrieving our `port` from `ports` on the event delivered to `onmessage()`. Then, we register an `onmessage` function *on that port*, which is how we receive the light levels that Java delivers via `postMessage()` on its `WebMessagePort`. When the user taps the label, we call a `pull()` function in JavaScript, that calls `postMessage()` on the `port`, supplying some string as a message (here, hardcoded as `"ping"` and ignored by our Java code):

```
<html>
<head>
<title>Android Light Sensor Demo</title>
<script language="javascript">
  function update_lux(lux) {
    document.getElementById("lux").innerHTML=lux;
  }

  function parse(json) {
    var result=JSON.parse(json);

    update_lux(result.lux);
  }

  var port;

  function pull() {
    port.postMessage("ping");
  }

  onmessage = function (e) {
    port = e.ports[0];

    port.onmessage = function (f) {
      parse(f.data);
    }
  }
</script>
</head>
<body>
```

```
<p><b><a onClick="pull()">Light Level</a>: </b><span id="lux">0.0</span> lux</p>
</body>
</html>
```

(from WebKit/SensorPort/app/src/main/assets/index.html)

If you run this sample on Android 6.0+, and you tap the "Light Level" label, you will get the light level, delivered by means of our `WebMessagePort`-based communications channel.

(**NOTE**: the author would like to thank Diego Torres Milano for his assistance in finding out how this stuff works).

Navigating the Waters

There is no navigation toolbar with the WebView widget. This allows you to use it in places where such a toolbar would be pointless and a waste of screen real estate. That being said, if you want to offer navigational capabilities, you can, but you have to supply the UI. WebView offers ways to perform garden-variety browser navigation, including:

- `reload()` to refresh the currently-viewed Web page
- `goBack()` to go back one step in the browser history, and `canGoBack()` to determine if there is any history to go back to
- `goForward()` to go forward one step in the browser history, and `canGoForward()` to determine if there is any history to go forward to
- `goBackOrForward()` to go backwards or forwards in the browser history, where negative numbers represent a count of steps to go backwards, and positive numbers represent how many steps to go forwards
- `canGoBackOrForward()` to see if the browser can go backwards or forwards the stated number of steps (following the same positive/negative convention as `goBackOrForward()`)
- `clearCache()` to clear the browser resource cache and `clearHistory()` to clear the browsing history

Settings, Preferences, and Options (Oh, My!)

With your favorite desktop Web browser, you have some sort of "settings" or "preferences" or "options" window. Between that and the toolbar controls, you can tweak and twiddle the behavior of your browser, from preferred fonts to the behavior of JavaScript.

Similarly, you can adjust the settings of your WebView widget as you see fit, via the `WebSettings` instance returned from calling the widget's `getSettings()` method.

There are lots of options on `WebSettings` to play with. Most appear fairly esoteric (e.g., `setFantasyFontFamily()`). However, here are some that you may find more useful:

- Control the font sizing via `setDefaultFontSize()` (to use a point size) or `setTextSize()` (to use constants indicating relative sizes like LARGER and SMALLEST)
- Control Web site rendering via `setUserAgent()`, so you can supply your own user agent string to make the Web server think you are a desktop browser, another mobile device (e.g., iPhone), or whatever. The settings you change are not persistent, so you should store them somewhere (such as via the Android preferences engine) if you are allowing your users to determine the settings, versus hard-wiring the settings in your application.

Security and Your WebView

More so than normal widgets, `WebView` opens up potential security issues, just as a Web browser could. If all you are doing is displaying your own content, the risks are minimal. If, on the other hand, you are displaying content from third parties, it is possible that their content is malicious in a way that can compromise your app's security, to your users' detriment.

Rogue JavaScript Risks

If you call `setJavaScriptEnabled(true)` on your `WebSettings`, you are allowing JavaScript code to be loaded and executed by `WebView`. In many cases, this is essential to get your content to render properly (e.g., the JavaScript is issuing AJAX calls). However, if you did not write the scripts, you do not know what they might be doing. If there are flaws in `WebView` — such as those discussed in the next sections — then your users may be at risk.

Even in the absence of such bugs, JavaScript can always:

- Consume so much CPU that it represents an attempt at a denial-of-service attack on the user's device
- Access anything the user enters into the Web page

- Access anything *you* enter into the Web page, using approaches as `javascript:` URLs or `evaluateJavaScript()`

The addJavascriptInterface() Bugs

Another way that rogue JavaScript can attack users is if you use `addJavascriptInterface()` to allow JavaScript code to call out to a Java object that you supply.

As was noted earlier in this chapter, when `addJavascriptInterface()` was introduced, there is this `@JavascriptInterface` annotation that we should apply to the methods we want JavaScript to be able to call on the object we supply via `addJavascriptInterface()`. This is because of a bug in the `addJavascriptInterface()` implementation, whereby on 4.1 and below *any* method on the Java object could be called by JavaScript. This includes methods like `getClass()`... which in turn would allow JavaScript to use `Class.forName()` to get at arbitrary stuff. This was used by various bits of malware.

Hence, using `addJavascriptInterface()` on Android 4.1 and below is rather risky, if you are loading arbitrary third-party JavaScript. If you have the means of examining that JavaScript (e.g., you are loading the scripts yourself), you might perform some simple scans of it to see if they appear to be doing anything unfortunate with your Java object that you injected into JavaScript via `addJavascriptInterface()`.

Worse, Android sometimes also injects its *own* objects, without our requesting them.

In particular, this security bug points out that, through Android 4.3, if users have enabled an accessibility service, Android automatically injects objects into `WebView`, using `addJavascriptInterface()`, named `accessibility` and `accessibilityTraversal`. So, even if you do not inject any objects yourself via `addJavascriptInterface()`, your `WebView` may be at risk. The security researchers who uncovered this attack vector suggest using `removeJavascriptInterface()` to specifically get rid of those objects.

The Same-Origin Policy Bug

Due to a bizarre bug in the parsing of URLs, it is possible for JavaScript code to violate the "same-origin policy" of a `WebView` on Android 4.3 and earlier.

Quoting Wikipedia from September 2014:

> the same-origin policy is an important concept in the web application security model. The policy permits scripts running on pages originating from the same site — a combination of scheme, hostname, and port number — to access each other's DOM with no specific restrictions, but prevents access to DOM on different sites... This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions. A strict separation between content provided by unrelated sites must be maintained on the client side to prevent the loss of data confidentiality or integrity.

All modern Web browsers implement the same-origin policy (SOP)... but there can be bugs. A security researcher disclosed that the original AOSP Browser application failed to implement the SOP properly, when a `javascript:` URL has a null byte before the `j` in `javascript`. And while the report was focused on the AOSP Browser app, the problem really lies with `WebView`.

To see this in action, load `https://commonsware.com/misc/sop-demo.html` in the AOSP Browser app on Android 4.3 or lower. This Web page consists of:

```
<html>
<head>
<title>WebView SOP Test</title>
</head>
<body>
<h1>WebView SOP Test</h1>
<iframe name="test" src="http://developer.android.com"></iframe>
<input type=button value="test"
onclick="window.open('\u0000javascript:alert(document.domain)','test')">
</body>
</html>
```

It is derived from a similar example found in the blog post outlining the security flaw.

In an SOP-compliant browser, clicking the button will have no effect. In the AOSP Browser app, clicking the button shows the domain name of the document in the `iframe`. And, loading this HTML into a `WebView` has the same effect.

Part of the `WebView` overhaul in Android 4.4 — replacing the original implementation with a new one backed by Chromium — had the effect of fixing this bug, whether intentionally or inadvertently.

There is no obvious mitigation approach for this bug, insofar as for the attack shown above, none of the callbacks on `WebViewClient` or `WebChromeClient` seem to allow us to intercept this URL before its JavaScript is executed. If you are loading HTML yourself from a third party, you might consider scanning that HTML for obvious signs of the attack (e.g., regular expression check for `\u0000javascript`), but that will be limited at best. Beyond that, try to limit the content in a `WebView` to be from only one origin, so that there is nothing for attackers to obtain via this bug.

Also note that the security researcher who found this bug has also found another SOP violation, suggesting that mitigation strategies may be impractical.

Android 8.0 WebView Changes

`WebView`, as Android developers use it, is really a facade API, as of Android 5.0. Previously, the `WebView` implementation was part of the Android framework. Nowadays, the implementation is delegated to the Android System WebView, which allows Google to update the `WebView` implementation without relying upon manufacturers to distribute firmware updates.

Hence, to some extent, `WebView` continuously changes, as the Android System WebView app gets updated a few times per year.

However, it is only in a new version of Android that Google changes the API or the general approach taken by a `WebView` implementation. In Android 8.0, two significant changes arrived: multi-process mode and support for banning of cleartext traffic.

Multi-Process Mode

Historically, `WebView` code ran in your app's regular process. Because we often load JavaScript into a `WebView`, having the `WebView` in our process would raise the risks of `WebView` bugs. If JavaScript could cause arbitrary code to execute in our own process, it would have whatever rights our own application code does, including everything for which we have runtime permissions.

Android 7.0 added a developer option for moving the `WebView` execution to a separate process, and that is now standard on Android 8.0. As the documentation describes it, "Web content is handled in a separate, isolated process from the containing app's process for enhanced security."

In principle, this should not require code changes. However, this is a significant architectural change, and so it is worthwhile to fully test your `WebView` usage on Android 8.0+, to make sure nothing breaks compared with that same code running on Android 7.1 or older devices.

Honors Cleartext Traffic Setting

If your app has a `targetSdkVersion` over 25, and in your network security configuration you banned cleartext traffic, `WebView` will honor that setting on Android 8.0+. On older devices, `WebView` ignores the network security configuration, and it appears that it still ignores the rest of the configuration (e.g., certificate pinning).

However, this allows you to block the accidental transmission of data over unencrypted channels. Most apps doing network I/O should ban cleartext traffic, at least for `debug` builds, to identify where they are accidentally using a plain `http` URL. Apps for which security is a priority might also ban cleartext traffic for `release` builds.

The `WebKit/BrowserSecure` sample project demonstrates this. It has a network security configuration XML file that bands cleartext traffic:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config cleartextTrafficPermitted="false">
    <trust-anchors>
      <certificates src="system" />
    </trust-anchors>
  </base-config>
</network-security-config>
```

(from WebKit/BrowserSecure/app/src/main/res/xml/net_security_config.xml)

It applies that configuration in the manifest via the `android:networkSecurityConfig` attribute on the `<application>` element:

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:networkSecurityConfig="@xml/net_security_config">
    <activity android:name=".BrowserDemo1">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
```

```
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

(from WebKit/BrowserSecure/app/src/main/AndroidManifest.xml)

The activity just loads up a plain HTTP-served Web page:

```
package com.commonsware.android.browser1;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class BrowserDemo1 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    browser=(WebView)findViewById(R.id.webkit);
    browser.loadUrl("http://www.andglobe.com");
  }
}
```

(from WebKit/BrowserSecure/app/src/main/java/com/commonsware/android/browser1/BrowserDemo1.java)

The project is set up to target 26:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 25
        targetSdkVersion 26
        applicationId 'com.commonsware.android.browser.secure'
    }
}
```

(from WebKit/BrowserSecure/app/build.gradle)

If you change that to have it target 25, the app will run "normally" on Android 8.0, showing the designated Web page. But, with a `targetSdkVersion` set at `26`, the `WebView` will not show that page, as it would be loaded over a cleartext (HTTP) connection:

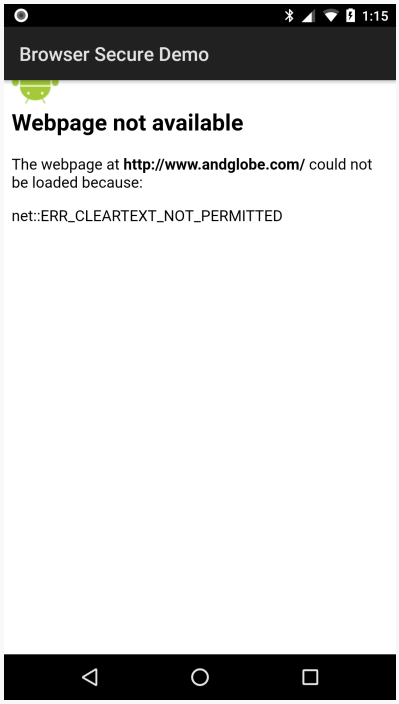


Figure 554: BrowserSecure Sample, Showing Banned Cleartext Traffic

Chrome Custom Tabs

Chrome custom tabs serve as middle ground between using a `WebView` in your own app and launching a URL into a separate Web browser.

With a `WebView`, you have complete control over the overall user experience within your app. However, your `WebView` is decoupled from any other browser the user may be using on the device. Conversely, launching URLs into the user's chosen browser gives the user their normal browsing experience, but you have no control over the user experience, as the user is now in the browser app, not your app.

With Chrome custom tabs, while Chrome is handling the URL, it will allow you limited control over the action bar (color and custom actions). It also simplifies some things that you might otherwise have had to handle yourself, such as pre-

fetching a Web page to be able to quickly switch to it. Basic integration is also fairly easy, coming in the form of extras on the same sort of `ACTION_VIEW Intent` that you might have used for launching the URL in a standalone browser.

At the same time, there are some concerns:

- The documentation states that there is a "Shared Cookie Jar and permissions model so users don't have to log in to sites they are already connected to, or re-grant permissions they have already granted", which would require significant testing to ensure that you are not leaking information into Chrome that might be somehow delivered to other sites (including Google).
- While it uses an `ACTION_VIEW Intent`, and so the user can choose to view the URL in a different browser, you will not get the custom integration in that case. This may be fine, but you will need to make sure that from a marketing and documentation standpoint you handle both the case where the user chooses Chrome (and you get the "custom tab") and the case where the user chooses something else.
- Since any app could handle the `ACTION_VIEW Intent`, you need to take into account that any information in the custom extras, like the `PendingIntent` to use for an custom action bar item, is stuff that you are willing for *arbitrary* apps to get their hands on. Do not assume that your communications will solely be with Chrome.

The Input Method Framework

We think of Android devices as having "soft keyboards". The official term for this is that Android devices offer one or more "input method editors" (or "input methods" for short). These input methods allow for text entry on a touchscreen, avoiding the need for a physical keyboard. Note, though, that "text entry" does not necessarily imply an on-screen keyboard equivalent — for example, the old PalmOS Graffiti text entry system is available as an app on the Play Store.

While it is possible to create custom input method editors — as the authors of Graffiti Pro did — this chapter is focused more on how ordinary app developers are affected by input methods, and how an app can help steer the behavior of the input method to benefit the user.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the section covering the `EditText` widget.

Keyboards, Hard and Soft

Some Android devices have a hardware keyboard that is visible some of the time (when it is slid out). A few Android devices have a hardware keyboard that is always visible (so-called "bar" or "slab" phones). Most Android devices, though, have no hardware keyboard at all.

The IMF handles all of these scenarios. In short, if there is no hardware keyboard, an input method editor (IME) will be available to the user when they tap on an enabled `EditText`.

This requires no code changes to your application... if the default functionality of the IME is what you want. Fortunately, Android is fairly smart about guessing what you want, so it may be you can just test with the IME but otherwise make no specific code changes.

Of course, the keyboard may not quite behave how you would like. For example, in the `Basic/Field` sample project, the `FieldDemo` activity has the IME overlaying the multiple-line `EditText`:

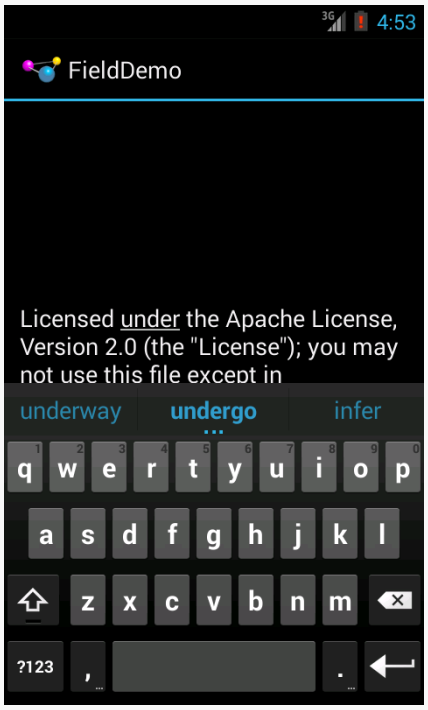


Figure 555: The input method editor, as seen in the FieldDemo sample application

It would be nice to have more control over how this appears, and for other behavior of the IME. Fortunately, the framework as a whole gives you many options for this, as is described over the bulk of this chapter.

Tailored To Your Needs

Android 1.1 and earlier offered many attributes on `EditText` widgets to control their style of input, such as `android:password` to indicate a field should be for password entry (shrouding the password keystrokes from prying eyes). Starting in Android 1.5, with the IMF, many of these have been combined into a single `android:inputType` attribute.

The `android:inputType` attribute takes a class plus modifiers, in a pipe-delimited list (where `|` is the pipe character). The class generally describes what the user is

allowed to input, and this determines the basic set of keys available on the soft keyboard. The available classes are:

1. `text` (the default)
2. `number`
3. `phone`
4. `datetime`
5. `date`
6. `time`

Many of these classes offer one or more modifiers, to further refine what the user will be entering. To help explain those, take a look at the `res/layout/main.xml` file from the `InputMethod/IMEDemo1` project:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView
      android:text="No special rules:"
    />
    <EditText
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Email address:"
    />
    <EditText
      android:inputType="text|textEmailAddress"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Signed decimal number:"
    />
    <EditText
      android:inputType="number|numberSigned|numberDecimal"
    />
  </TableRow>
  <TableRow>
    <TextView
```

```
      android:text="Date:"
    />
    <EditText
      android:inputType="date"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Multi-line text:"
    />
    <EditText
      android:inputType="text|textMultiLine|textAutoCorrect"
      android:minLines="3"
      android:gravity="top"
    />
  </TableRow>
</TableLayout>
```

(from InputMethod/IMEDemo1/app/src/main/res/layout/main.xml)

Here, you will see a `TableLayout` containing five rows, each demonstrating a slightly different flavor of `EditText`:

- One has no attributes at all on the `EditText`, meaning you get a plain text entry field
- One has `android:inputType = "text|textEmailAddress"`, meaning it is text entry, but specifically seeks an email address
- One allows for signed decimal numeric input, via `android:inputType = "number|numberSigned|numberDecimal"`
- One is set up to allow for data entry of a date (`android:inputType = "date"`)
- The last allows for multi-line input with auto-correction of probable spelling errors (`android:inputType = "text|textMultiLine|textAutoCorrect"`)

The class and modifiers tailor the keyboard. So, a plain text entry field results in a plain soft keyboard:

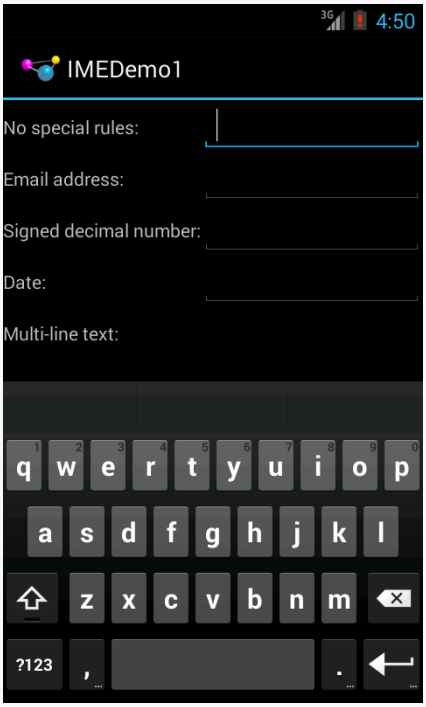


Figure 556: A standard input method editor (a.k.a., soft keyboard)

An email address field might put the `@` symbol on the soft keyboard, perhaps at the cost of a smaller spacebar, depending on the keyboard implementation:

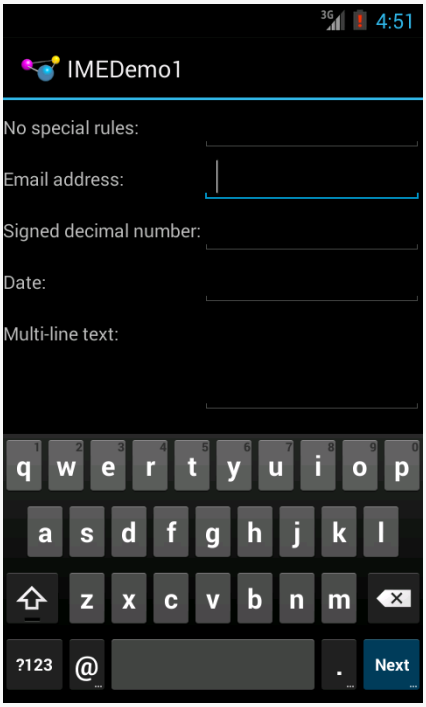


Figure 557: The input method editor for email addresses

Note, though, that this behavior is specific to the input method editor. Some editors might put an @ sign on the primary keyboard for an email field. Some might put a ".com" button on the primary keyboard. Some might not react at all. It is up to the implementation of the input method editor — all you can do is supply the hint.

Numbers and dates restrict the keys to numeric keys, plus a set of symbols that may or may not be valid on a given field:

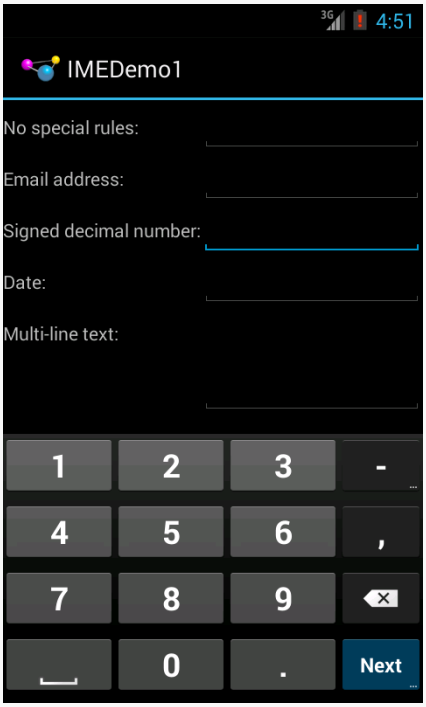


Figure 558: The input method editor for signed decimal numbers

And so on.

By choosing the appropriate `android:inputType`, you can give the user a soft keyboard that best suits what it is they should be entering.

Tell Android Where It Can Go

You may have noticed a subtle difference between the first and second input method editors, beyond the addition of the `@` key. If you look in the lower-right corner of the soft keyboard, the second field's editor has a "Next" button, while the first field's editor has a newline button.

This points out two things:

- `EditText` widgets are multi-line by default if you do not specify `android:inputType`
- You can control what goes on with that lower-right-hand button, called the action key

By default, on an `EditText` where you have specified `android:inputType`, the action key will be "Next", moving you to the next `EditText` in sequence, or "Done", if you are on the last `EditText` on the screen. You can manually stipulate what the action key will be labeled via the `android:imeOptions` attribute. For example, in the `res/layout/main.xml` from the InputMethod/IMEDemo2 sample project, you will see an augmented version of the previous example, where two input fields specify what their action key should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
>
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
    >
    <TableRow>
      <TextView
        android:text="No special rules:"
      />
      <EditText
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Email address:"
      />
      <EditText
        android:inputType="text|textEmailAddress"
        android:imeOptions="actionSend"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Signed decimal number:"
      />
      <EditText
        android:inputType="number|numberSigned|numberDecimal"
        android:imeOptions="actionDone"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Date:"
```

```
      />
      <EditText
        android:inputType="date"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Multi-line text:"
      />
      <EditText
        android:inputType="text|textMultiLine|textAutoCorrect"
        android:minLines="3"
        android:gravity="top"
      />
    </TableRow>
  </TableLayout>
</ScrollView>
```

(from InputMethod/IMEDemo2/app/src/main/res/layout/main.xml)

Here, we attach a "Send" action to the action key for the email address (`android:imeOptions = "actionSend"`), and the "Done" action on the middle field (`android:imeOptions = "actionDone"`).

By default, "Next" will move the focus to the next `EditText` and "Done" will close up the input method editor. However, for those, or for any other ones like "Send", you can use `setOnEditorActionListener()` on `EditText` (technically, on the `TextView` superclass) to get control when the action key is clicked or the user presses the `<Enter>` key. You are provided with a flag indicating the desired action (e.g., `IME_ACTION_SEND`), and you can then do something to handle that request (e.g., send an email to the supplied email address).

If you need more control over the action button, you can set:

- `android:imeActionId`, which provides a custom value for the `actionId` that is passed to `onEditorAction()` of your `OnEditorActionListener`
- `android:imeActionLabel`, where you provide your own caption for the button (bearing in mind that your desired caption may or may not fit)

Fitting In

You will notice that the `IMEDemo2` layout shown above has another difference from its `IMEDemo1` predecessor: the use of a `ScrollView` container wrapping the `TableLayout`.

This ties into another level of control you have over the input method editors: what happens to your activity's own layout when the input method editor appears?

There are three possibilities, depending on circumstances:

1. Android can "pan" your activity, effectively sliding the whole layout up to accommodate the input method editor, or overlaying your layout, depending on whether the `EditText` being edited is at the top or bottom. This has the effect of hiding some portion of your UI.
2. Android can resize your activity, effectively causing it to shrink to a smaller screen dimension, allowing the input method editor to sit below the activity itself. This is great when the layout can readily be shrunk (e.g., it is dominated by a list or multi-line input field that does not need the whole screen to be functional).
3. In landscape mode, Android may display the input method editor full-screen, obscuring your entire activity. This allows for a bigger keyboard and generally easier data entry.

Android controls the full-screen option purely on its own. And, by default, Android will choose between pan and resize modes depending on what your layout looks like. If you want to specifically choose between pan and resize, you can do so via an `android:windowSoftInputMode` attribute on the `<activity>` element in your `AndroidManifest.xml` file. For example, here is the manifest from `IMEDemo2`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.imf.two"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
```

```
      android:name=".IMEDemo2"
      android:label="@string/app_name"
      android:windowSoftInputMode="adjustResize">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from InputMethod/IMEDemo2/app/src/main/AndroidManifest.xml)

Because we specified resize, Android will shrink our layout to accommodate the input method editor. With the `ScrollView` in place, this means the scrollbar will appear as needed when the user is scrolling:

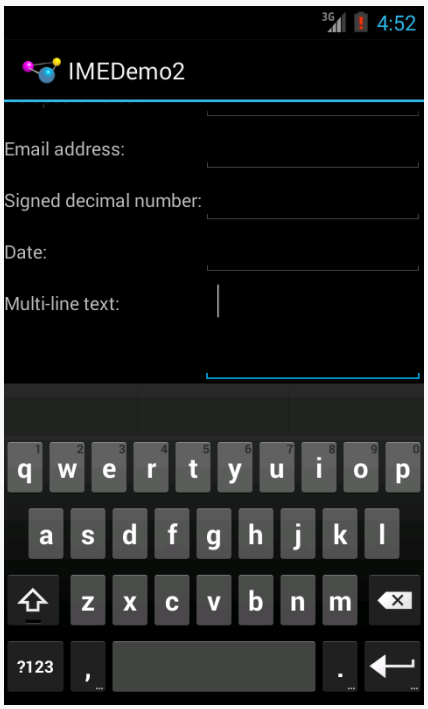


Figure 559: The shrunken, scrollable layout

Jane, Stop This Crazy Thing!

Sometimes, you need the input method editor to just go away. For example, if you make the action button be “Search”, the user tapping that button will not automatically hide the editor.

To hide the editor, you will need to make a call to the `InputMethodManager`, a system service that controls these input method editors:

```
InputMethodManager
mgr=(InputMethodManager)getSystemService(INPUT_METHOD_SERVICE);

mgr.hideSoftInputFromWindow(fld.getWindowToken(), 0);
```

(where `fld` is the `EditText` whose input method editor you want to hide)

Fonts and Text

Inevitably, you'll get the question "hey, can we change this font?" when doing application development. The answer depends on what fonts come with the platform, whether you can add other fonts, and how to apply them to the widget or whatever needs the font change.

Android is no different. It comes with some fonts plus a means for adding new fonts. Though, as with any new environment, there are a few idiosyncrasies to deal with.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on files.

Love The One You're With

Android natively knows three fonts, by the shorthand names of "sans", "serif", and "monospace". For Android 1.x, 2.x, and 3.x, these fonts are actually the Droid series of fonts, created for the Open Handset Alliance by Ascender. A new font set, Roboto, is used in Android 4.x and beyond, though the look of the font changed somewhat in Android 5.0.

For those fonts, you can just reference them in your layout XML, if you choose, such as the following layout from the `Fonts/FontSampler` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
```

```
android:stretchColumns="1">

<TableRow>

  <TextView
    android:layout_marginRight="4dip"
    android:text="@string/label_sans"
    android:textSize="20sp" />

  <TextView
    android:id="@+id/sans"
    android:text="@string/hello_world"
    android:textSize="20sp"
    android:typeface="sans" />
</TableRow>

<TableRow>

  <TextView
    android:layout_marginRight="4dip"
    android:text="@string/label_serif"
    android:textSize="20sp" />

  <TextView
    android:id="@+id/serif"
    android:text="@string/hello_world"
    android:textSize="20sp"
    android:typeface="serif" />
</TableRow>

<TableRow>

  <TextView
    android:layout_marginRight="4dip"
    android:text="@string/label_monospace"
    android:textSize="20sp" />

  <TextView
    android:id="@+id/monospace"
    android:text="@string/hello_world"
    android:textSize="20sp"
    android:typeface="monospace" />
</TableRow>

<TableRow>

  <TextView
    android:layout_marginRight="4dip"
```

```
      android:text="@string/label_custom"
      android:textSize="20sp" />

    <TextView
      android:id="@+id/custom"
      android:text="@string/hello_world"
      android:textSize="20sp" />
  </TableRow>

  <TableRow android:id="@+id/filerow">

    <TextView
      android:layout_marginRight="4dip"
      android:text="@string/label_custom_from_file"
      android:textSize="20sp" />

    <TextView
      android:id="@+id/file"
      android:text="@string/hello_world"
      android:textSize="20sp" />
  </TableRow>
</TableLayout>
```

(from Fonts/FontSampler/app/src/main/res/layout/main.xml)

This layout builds a table showing short samples of five fonts. Notice how the first three have the `android:typeface` attribute, whose value is one of the three built-in font faces (e.g., “sans”).

The three built-in fonts are very nice. However, it may be that a designer, or a manager, or a customer wants a different font than one of those three. Or perhaps you want to use a font for specialized purposes, such as an image font instead of a series of PNG graphics.

The easiest way to accomplish this is to package the desired font(s) with your application. To do this, simply create an `assets/` folder in the project root, and put your TrueType (TTF) fonts in the assets. You might, for example, create `assets/fonts/` and put your TTF files in there. Note that Android has some support for OpenType (OTF) fonts, as well.

Then, you need to tell your widgets to use that font. Unfortunately, you can no longer use layout XML for this, since the XML does not know about any fonts you may have tucked away as an application asset. Instead, you need to make the change in Java code:

```
package com.commonsware.android.fonts;

import android.app.Activity;
import android.graphics.Typeface;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.TextView;
import java.io.File;

public class FontSampler extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    TextView tv=(TextView)findViewById(R.id.custom);
    Typeface face=
      Typeface.createFromAsset(getAssets(), "fonts/HandmadeTypewriter.ttf");

    tv.setTypeface(face);

    File font=
      new File(Environment.getExternalStorageDirectory(), "MgOpenCosmeticaBold.ttf");

    if (font.exists()) {
      tv=(TextView)findViewById(R.id.file);
      face=Typeface.createFromFile(font);

      tv.setTypeface(face);
    }
    else {
      findViewById(R.id.filerow).setVisibility(View.GONE);
    }
  }
}
```

(from Fonts/FontSampler/app/src/main/java/com/commonsware/android/fonts/FontSampler.java)

Here we grab the `TextView` for our "custom" sample, then create a `Typeface` object via the static `createFromAsset()` builder method. This takes the application's `AssetManager` (from `getAssets()`) and a path within your `assets/` directory to the font you want.

Then, it is just a matter of telling the `TextView` to `setTypeface()`, providing the `Typeface` you just created. In this case, we are using the Handmade Typewriter font.

You can also load a font out of a local file and use it. The benefit is that you can customize your fonts after your application has been distributed. On the other hand, you have to somehow arrange to get the font onto the device. But just as you can get a `Typeface` via `createFromAsset()`, you can get a `Typeface` via `createFromFile()`. In our `FontSampler`, we look in the root of "external storage" (typically the SD card) for

the MgOpenCosmeticaBold TrueType font file, and if it is found, we use it for the fifth row of the table. Otherwise, we hide that row.

The results?

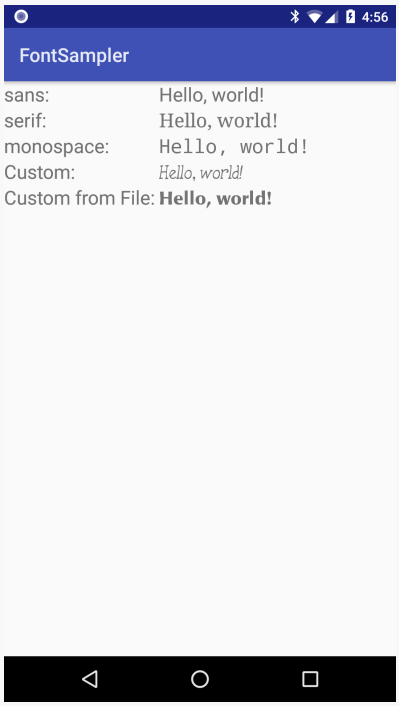


Figure 560: The FontSampler application

Note that Android does not seem to like all TrueType fonts. When Android dislikes a custom font, rather than raise an `Exception`, it seems to substitute Droid Sans ("sans") quietly. So, if you try to use a different font and it does not seem to be working, it may be that the font in question is incompatible with Android, for whatever reason.

Yeah, But Do We Really Have To Do This in Java?

One common complaint with font handling in Android is that you have to apply a custom font on a per-widget basis in Java code.

This gets old quickly.

It is not too bad with just a single `TextView`. But for a whole activity, or a whole application, changing all of the relevant `TextView` widgets (and descendents, like `Button`) gets to be a bit tedious.

While there are "traverse the widget hierarchy and fix up the fonts" code snippets available, you are probably better served using a third-party library, like Christoper Jenkins' Calligraphy, which lets you define custom fonts in layout XML files or style resources.

Here a Glyph, There a Glyph

TrueType fonts can be rather pudgy, particularly if they support an extensive subset of the available Unicode characters. The Handmade Typewriter font used above runs over 70KB; the DejaVu free fonts can run upwards of 500KB apiece. Even compressed, these add bulk to your application, so be careful not to go overboard with custom fonts, lest your application take up too much room on your users' phones.

Conversely, bear in mind that fonts may not have all of the glyphs that you need. As an example, let us talk about the ellipsis.

Android's `TextView` class has the built-in ability to "ellipsize" text, truncating it and adding an ellipsis if the text is longer than the available space. You can use this via the `android:ellipsize` attribute, for example. This works fairly well, at least for single-line text.

The ellipsis that Android uses is not three periods. Rather it uses an actual ellipsis character, where the three dots are contained in a single glyph. Hence, any font that you use in a `TextView` where you also use the "ellipsizing" feature will need the ellipsis glyph.

Beyond that, though, Android pads out the string that gets rendered on-screen, such that the length (in characters) is the same before and after "ellipsizing". To make this work, Android replaces one character with the ellipsis, and replaces all other removed characters with the Unicode character 'ZERO WIDTH NO-BREAK SPACE' (`U+FEFF`). This means the "extra" characters after the ellipsis do not take up any visible space on screen, yet they can be part of the string.

However, this means any custom fonts you use for `TextView` widgets that you use with `android:ellipsize` must also support this special Unicode character. Not all

fonts do, and you will get artifacts in the on-screen representation of your shortened strings if your font lacks this character (e.g., rogue X's appear at the end of the line).

And, of course, Android's international deployment means your font must handle any language your users might be looking to enter, perhaps through a language-specific input method editor.

Hence, while using custom fonts in Android is very possible, there are many potential problems, and so you must weigh carefully the benefits of the custom fonts versus their potential costs.

Auto-Sizing TextView

Text comes in all lengths.

Sometimes, when showing text in a `TextView`, we can allow that text to word-wrap and extend vertically. Other times, though, that proves to be impractical, such as when using a `TextView` as a label for another widget. However, even in those times, the text to be shown may vary in length by a significant amount. Translations of the label's text might range from a couple of kanji in Japanese to a 20-letter word in German or Icelandic.

One long-standing solution to that problem has been to use an auto-sizing `TextView`. Here, what is being "auto-sized" is the size of the font used for the text, to keep the `TextView` at a fixed dimension regardless of translation. It is up to the developer to ensure that for a given screen size and translation that the font does not wind up being too small to be read.

However, Android itself never had a widget for this, so developers would rely instead upon third-party or home-grown implementations.

Android 8.0 adds this sort of auto-sizing capability to `TextView` itself (and, by extension, subclasses like `Button`). You have two main approaches for implementing this:

- Set `android:autoSizeTextType="uniform"` on the `TextView` to engage auto-size capability. Then, use `android:autoSizeMinTextSize` and `android:autoSizeMaxTextSize` to set the lower and upper bounds for the text size, where Android can choose any size in between those. If you also add `android:autoSizeStepGranularity`, you can choose the increments in

which Android will move between the ends of the text size range (the default is 1px).

- Set android:autoSizeTextType="uniform" on the TextView, as in the above option. Then, instead of providing the starting and ending values of a range, use an <array> resource to define specific sizes that you want to support, then use android:autoSizePresetSizes to point to that resource. This is a bit more difficult to set up, as you need to define the array resource, but it may simplify testing, as you are in complete control over the possible sizes.

To make this work, you need to constrain the size of the TextView, so that it does not expand to fill all available space. That might be through having both axes set to wrap_content, but have the container holding the TextView limit how big the TextView can get. It might be through TextView-specific configuration, such as android:maxLines. Or, it might be through setting the size of the axes to be some specific dimension or match_parent.

The Basic/AutoSize sample project illustrates the use of both forms of android:autoSizeTextType.

The activity's layout consists of an EditText, a pair of TextView widgets, and a pair of divider lines (horizontal View widgets with background and margin), all inside of a vertical LinearLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context="com.commonsware.android.autosize.MainActivity">

  <EditText
    android:id="@+id/input"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:hint="@string/hint_input" />

  <View
    android:layout_width="match_parent"
    android:layout_height="2dp"
    android:layout_margin="4dp"
    android:background="@android:color/black" />
```

```
  <TextView
    android:id="@+id/granular"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:autoSizeMaxTextSize="40sp"
    android:autoSizeMinTextSize="5sp"
    android:autoSizeStepGranularity="5sp"
    android:autoSizeTextType="uniform"
    android:maxLines="1" />

  <View
    android:layout_width="match_parent"
    android:layout_height="2dp"
    android:layout_margin="4dp"
    android:background="@android:color/black" />

  <TextView
    android:id="@+id/steps"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:autoSizePresetSizes="@array/autosize_sizes"
    android:autoSizeTextType="uniform"
    android:maxLines="1" />

</LinearLayout>
```

(from Basic/AutoSize/app/src/main/res/layout/activity_main.xml)

Both `TextView` widgets have `android:autoSizeTextType` set to `uniform`. The top one uses `android:autoSizeMinTextSize`, `android:autoSizeMaxTextSize`, and `android:autoSizeStepGranularity` to allow the text size to float between `5sp` and `40sp` in `5sp` increments. The bottom one uses `android:autoSizePresetSizes` to tie in an `array` resource for the valid sizes to use:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <array name="autosize_sizes">
    <item>10sp</item>
    <item>12sp</item>
    <item>14sp</item>
    <item>16sp</item>
    <item>18sp</item>
    <item>20sp</item>
  </array>
</resources>
```

(from Basic/AutoSize/app/src/main/res/values/arrays.xml)

Here, the size can float between `10sp` and `20sp` in `2sp` increments. However, there is no requirement that the sizes increment in a uniform fashion when using the `array` approach.

The `MainActivity` that uses the layout sets up a `TextWatcher` on the `EditText` and copies what you enter into the two `TextView` widgets, to allow you to experiment in real time with changes in the text size:

```
    EditText input=(EditText)findViewById(R.id.input);
    final TextView granular=(TextView)findViewById(R.id.granular);
    final TextView steps=(TextView)findViewById(R.id.steps);

    input.addTextChangedListener(new TextWatcher() {
      @Override
      public void beforeTextChanged(CharSequence charSequence, int i, int i1,
                                    int i2) {
        // unused
      }

      @Override
      public void onTextChanged(CharSequence charSequence, int i, int i1,
                                int i2) {
        // unused
      }

      @Override
      public void afterTextChanged(Editable editable) {
        granular.setText(editable.toString());
        steps.setText(editable.toString());
      }
    });
  }
}
```

(from Basic/AutoSize/app/src/main/java/com/commonsware/android/autosize/MainActivity.java)

When you run the app, initially the `EditText` and corresponding `TextView` widgets have no text. If you start typing, you will see your text appear in the `TextView` widgets:

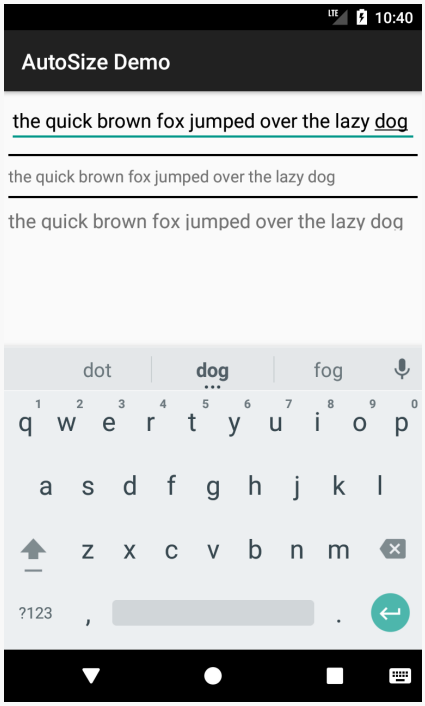


Figure 561: Auto-Sizing TextViews, with Some Text

As you continue typing, the text size decreases:

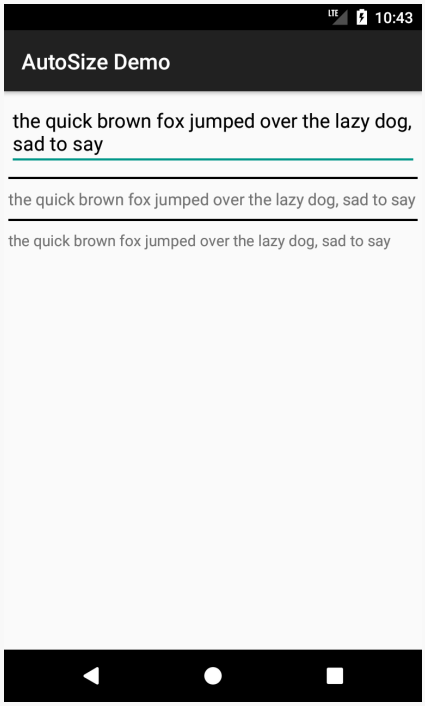


Figure 562: Auto-Sizing TextViews, with Some More Text

If you are using `appcompat-v7`, there are equivalent capabilities added to its wrapper backport of `TextView`.

Justified Text

If you are reading the PDF edition of this book, its paragraphs do not use justified text. If you are reading the Kindle edition of this book — at least when using the Amazon Kindle app for Android — its paragraphs do use justified text.

Justified text is when text (e.g., in the middle of a paragraph) fills the entire line, usually with bits of extra space added between the words to get them to neatly fill the available horizontal space.

For years, `TextView` never supported justified text, though you could get it through third-party libraries.

In Android 8.0+, you can now enable justified text on a `TextView` via `setJustify()`. At the present time, the documentation does not show an `android:justify` attribute, though, with luck, it will be added in a future update.

Rich Text

Plain text is so, well, plain.

Fortunately, Android has fairly extensive support for formatted text, before you need to break out something as heavy-weight as `WebView`. However, some of this rich text support has been shrouded in mystery, particularly how you would allow users to edit formatted text.

This chapter will explain how the rich text support in Android works and how you can take advantage of it, with particular emphasis on some open source projects to help you do just that.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic widgets and the input method framework.

The Span Concept

You may have noticed that many methods in Android accept or return a `CharSequence`. The `CharSequence` interface is little used in traditional Java, if for no other reason than there are relatively few implementations of it outside of `String`. However, in Android, `CharSequence` becomes much more important, because of a sub-interface named `Spanned`.

`Spanned` defines sequences of characters (`CharSequence`) that contain *inline markup rules*. These rules — mostly instances of `CharacterStyle` and `ParagraphStyle` subclasses – indicate whether the "spanned" portion of the characters should be

rendered in an alternate font, or be turned into a hyperlink, or have other effects applied to them.

Methods that take a `CharSequence` as a parameter, therefore, can work equally well with `String` objects as well as objects that implement `Spanned`.

Implementations

The base interface for rich-text `CharSequence` objects is `Spanned`. This is used for any `CharSequence` that has inline markup rules, and it defines methods for retrieving markup rules applied to portions of the underlying text.

The primary concrete implementation of `Spanned` is `SpannedString`. `SpannedString`, like `String`, is immutable — you cannot change either the text or the formatting of a `SpannedString`.

There is also the `Spannable` sub-interface of `Spanned`. `Spannable` is used for any `CharSequence` with inline markup rules that can be modified, and it defines the methods for modifying the formatting. There is a corresponding `SpannableString` implementation.

Finally, there is a related `Editable` interface, which is for a `CharSequence` that can have its *text* modified in-place. `SpannableStringBuilder` implements both `Editable` and `Spannable`, for modifying text and formatting at the same time.

TextView and Spanned

One of the most important uses of `Spanned` objects is with `TextView`. `TextView` is capable of rendering a `Spanned`, complete with all of the specified formatting. So, if you have a `Spanned` that indicates that the third word should be rendered in italics, `TextView` will faithfully italicize that word.

`TextView`, of course, is an ancestor of many other widgets, from `EditText` to `Button` to `CheckBox`. Each of those, therefore, can use and render `Spannable` objects. The fact that `EditText` has the ability to render `Spanned` objects — and even allow them to be edited — is key for allowing users to enter rich text themselves as part of your UI.

Available Spans

As noted above, the markup rules come in the form of instances of base classes known as `CharacterStyle` and `ParagraphStyle`. Despite those names, most of the SDK-supplied subclasses of `CharacterStyle` and `ParagraphStyle` end in `Span` (not `Style`), and so you will likely see references to these as "spans" as often as "styles". That also helps minimize confusion between character styles and style resources.

There are well over a dozen supplied `CharacterStyle` subclasses, including:

1. `ForegroundColorSpan` and `BackgroundColorSpan` for coloring text
2. `StyleSpan`, `TextAppearanceSpan`, `TypefaceSpan`, `UnderlineSpan`, and `StrikethroughSpan` for affecting the true "style" of text
3. `AbsoluteSizeSpan`, `RelativeSizeSpan`, `SuperscriptSpan`, and `SubscriptSpan` for affecting the size (and, in some cases, vertical position) of the text

And so on. Similarly, `ParagraphStyle` has subclasses like `BulletSpan` for bulleted lists.

You can implement your own custom subclasses of `CharacterStyle` and `ParagraphStyle`, though the book does not cover this subject at this time.

Loading Rich Text

`Spanned` objects do not appear by magic. Plenty of things in Java will give you ordinary strings, from XML and JSON parsers to loading data out of a database to simply hard-coding string constants. However, there are only a few ways that you as a developer will get a `Spanned` complete with formatting, and that includes you creating such a `Spanned` yourself by hand.

String Resource

The primary way most developers get a `Spanned` object into their application is via a string resource. String resources support inline markup in the form of HTML tags. Bold (`<b>`), italics (`<i>`), and underline (`<u>`) are officially supported, such as:

```
<string name="welcome">Welcome to <b>Android</b>!</string>
```

When you retrieve the string resource via `getText()`, you get back a `CharSequence` that represents a `Spanned` object with the markup rules in place.

HTML

The next-most common way to get a `Spanned` object is to use `Html.fromHtml()`. This parses an HTML string and returns a `Spanned` object, with all recognized tags converted into corresponding spans. You might use this for text loaded from a database, retrieved from a Web service call, extracted from an RSS feed, etc.

Unfortunately, the list of tags that `fromHtml()` understands is undocumented. Based upon the source code to `fromHtml()`, the following seem safe:

- `<a href="...">`
- `<b>`
- `<big>`
- `<blockquote>`
- `<br>`
- `<cite>`
- `<dfn>`
- `<div align="...">`
- `<em>`
- `<font size="..." color="..." face="...">`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<i>`
- `<img src="...">`
- `<p>`
- `<small>`
- `<strong>`
- `<sub>`
- `<sup>`
- `<tt>`
- `<u>`

However, do bear in mind that these are undocumented and therefore are subject to change. Also note that `fromHtml()` is perhaps slower than you might think, particularly for longer strings.

You might also wind up using some other support code to get your HTML. For example, some data sources might publish text formatted as Markdown — Stack Overflow, GitHub, etc. use this extensively. Markdown can be converted to HTML, through any number of available Java libraries or via CWAC-AndDown, which wraps the native hoedown Markdown-to-HTML converter for maximum speed. CWAC-AndDown will be explored in a bit more detail in the chapter on the NDK.

From EditText

The reason why so much sample code calls `getText()` followed by `toString()` on an `EditText` widget is because `EditText` is going to return an `Editable` object from `getText()`, not a simple string. That's because, in theory, `EditText` could be returning something with formatting applied. The call to `toString()` simply strips out any potential formatting as part of giving you back a `String`.

However, you could elect to use the `Editable` object (presumably a `SpannableStringBuilder`) if you wanted, such as for pouring the entered text into a `TextView`, complete with any formatting that might have wound up on the entered text.

Manually

You are welcome to create a `SpannableString` via its constructor, supplying the text that you wish to display, then calling various methods on `SpannableString` to format it.

Or, you are welcome to create a `SpannableStringBuilder` via its constructor. In some respects, `SpannableStringBuilder` works like the classic `StringBuilder` — you call `append()` to add more text. However, `SpannableStringBuilder` also offers `delete()`, `insert()`, and `replace()` methods to modify portions of the existing content. It also supports the same methods that `SpannableString` does, via the `Spannable` interface, for applying formatting rules to portions of text.

Editing Rich Text

If the Spannable you wound up with is a SpannedString, it is what it is — you cannot change it. If, however, you have a SpannableString, that can be modified by you, or by the user. Of course, allowing the user to modify a Spannable gets a wee bit tricky, and is why the RichEditText project was born.

Spannable offers two methods for modifying its formatting: setSpan() to apply formatting, and removeSpan() to get rid of an existing span. And, since Spannable extends Spanned, a Spannable also has getSpans(), to return existing spans of a current type within a certain range of characters in the text. These methods, along with others on Spanned, allow you to get and set whatever formatting you wish to apply on a Spannable object, such as a SpannableString.

For example, let's take a look at the RichText/Search sample project. Here, we are going to load some text into a TextView, then allow the user to enter a search string in an EditText, and we will use the Spannable methods to highlight the search string occurrences inside the text in the TextView.

Our layout is simply an EditText atop a TextView (wrapped in a ScrollView):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <EditText
    android:id="@+id/search"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:singleLine="true">

    <requestFocus/>
  </EditText>

  <ScrollView
    android:id="@+id/scroll"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
      android:id="@+id/prose"
      android:layout_width="match_parent"
```

```
      android:layout_height="wrap_content"
      android:text="@string/address"
      android:textAppearance="?android:attr/textAppearanceMedium"/>
  </ScrollView>

</LinearLayout>
```

(from RichText/Search/app/src/main/res/layout/main.xml)

We pre-fill the `TextView` with a string resource (`@string/address`), which in this project is the text of Lincoln's Gettysburg Address, with a bit of inline markup (e.g., "Four score and seven years ago" italicized). So, when we fire up the project at the outset, we see the formatted prose from the string resource:

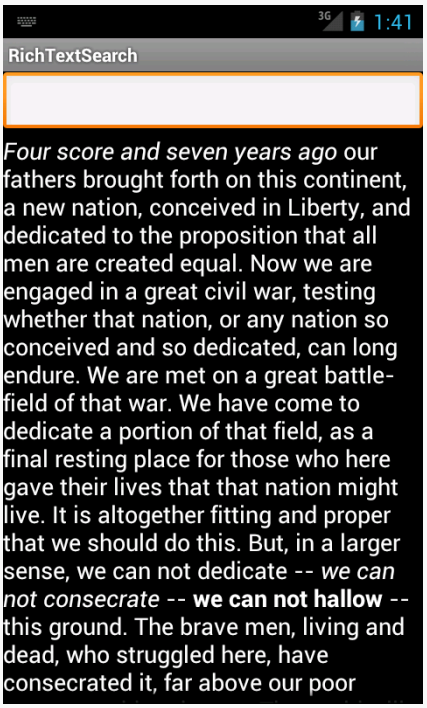


Figure 563: The RichTextSearch sample, as initially launched

In `onCreate()` of our activity, we find the `EditText` widget and designate the activity itself as being an `OnEditorActionListener` for the `EditText`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
```

```
    search=(EditText)findViewById(R.id.search);
    search.setOnEditorActionListener(this);
  }
```

(from RichText/Search/app/src/main/java/com/commonsware/android/rich/search/RichTextSearchActivity.java)

That means when the user presses <Enter>, we will get control in an `onEditorAction()` method. There, we pass the search text to a private `searchFor()` method, plus ensure that the input method editor is hidden (if one was used to fill in the search text):

```
  @Override
  public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
    if (event == null || event.getAction() == KeyEvent.ACTION_UP) {
      searchFor(search.getText().toString());

      InputMethodManager imm=
          (InputMethodManager)getSystemService(INPUT_METHOD_SERVICE);

      imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
    }

    return(true);
  }
```

(from RichText/Search/app/src/main/java/com/commonsware/android/rich/search/RichTextSearchActivity.java)

The `searchFor()` method is where the formatting is applied to our search text:

```
  private void searchFor(String text) {
    TextView prose=(TextView)findViewById(R.id.prose);
    Spannable raw=new SpannableString(prose.getText());
    BackgroundColorSpan[] spans=raw.getSpans(0,
                                             raw.length(),
                                             BackgroundColorSpan.class);

    for (BackgroundColorSpan span : spans) {
      raw.removeSpan(span);
    }

    int index=TextUtils.indexOf(raw, text);

    while (index >= 0) {
      raw.setSpan(new BackgroundColorSpan(0xFF8B008B), index, index
          + text.length(), Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
      index=TextUtils.indexOf(raw, text, index + text.length());
    }
```

```
    prose.setText(raw);
  }
```

(from RichText/Search/app/src/main/java/com/commonsware/android/rich/search/RichTextSearchActivity.java)

First, we get a `Spannable` object out of the `TextView`. While an `EditText` returns an `Editable` from `getText()`, `getText()` on a `TextView` returns a `CharSequence`. In particular, the first time we execute `searchFor()`, `getText()` will return a `SpannedString`, as that is what a string resource turns into. However, that is not modifiable, so we convert it into a `SpannableString` so we can apply formatting to it. An optimization would be to see if `getText()` returns something implementing `Spannable` and then just using it directly.

We want to highlight the search terms using a `BackgroundColorSpan`. However, that means we first need to get rid of any existing `BackgroundColorSpan` objects applied to the prose from a previous search — otherwise, we would keep highlighting more and more of the prose. So, we use `getSpans()` to find all `BackgroundColorSpan` objects anywhere in the prose (from index 0 through the length of the text). For each that we find, we call `removeSpan()` to get rid of it from our `Spannable`.

Then, we use `indexOf()` on `TextUtils` to find the first occurrence of whatever the user typed into the `EditText`. If we find it, we create a new `BackgroundColorSpan` and apply it to the matching portion of the prose using `setSpan()`. The last parameter to `setSpan()` is a flag, indicating what should happen if text is inserted at either the starting or ending point. In our case, the text itself is remaining constant, so the flag does not matter much – here, we use `SPAN_EXCLUSIVE_EXCLUSIVE`, which would mean that the span would not cover any text inserted at the starting or ending point of the span.

We then continue using `indexOf()` to find any remaining occurrences of the search text. Once we are done modifying our `Spannable`, we put it into the `TextView` via `setText()`.

The result is that all matching substrings are highlighted in a purple/magenta shade:

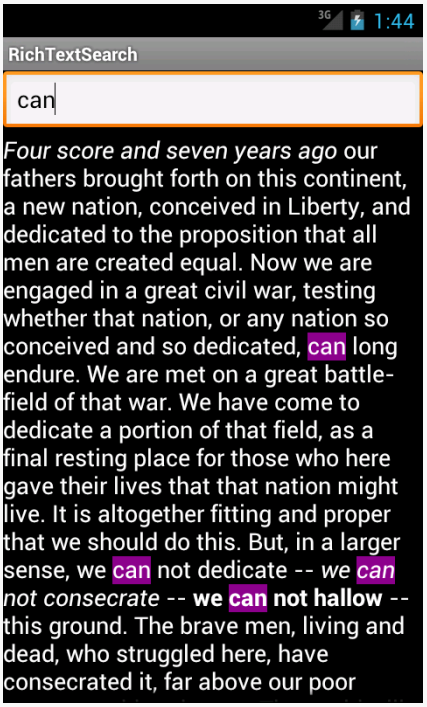


Figure 564: The RichTextSearch sample, after searching on "can"

Saving Rich Text

`SpannableString` and `SpannedString` are not `Serializable`. There is no built-in way to persist them directly.

However, `Html.toHtml()` will convert a `Spanned` object into corresponding HTML, for all `CharacterStyle` and `ParagraphStyle` objects that can be readily converted into HTML. You can then persist the resulting HTML any place you would persist a `String` (e.g., database column).

In principle, you could create other similar conversion code, such as something to take a `Spanned` and return the corresponding Markdown source.

Manipulating Rich Text

The `TextUtils` class has many utility methods that manipulate a `CharSequence`, to allow you to do things that you might ordinarily have done just with methods on `String`. These utility methods will work with any `CharSequence`, including `SpannedString` and `SpannableString`.

Some are specifically aimed at `Spanned` objects, such as `copySpansFrom()` (to apply formatting from one `CharSequence` onto another). Some are clones of `String` equivalents, such as `split()`, `join()`, and `substring()`. Yet others are designed for developers using the `Canvas` 2D drawing API, such as `ellipsize()` and `commaEllipsize()` for intelligently truncating messages.

Animators

Users like things that move. Or fade, spin, or otherwise offer a dynamic experience.

Much of the time, such animations are handled for us by the framework. We do not have to worry about sliding rows in a `ListView` when the user scrolls, or as the user pans around a `ViewPager`, and so forth.

However, sometimes, we will need to add our own animations, where we want effects that either are not provided by the framework innately or are simply different (e.g., want something to slide off the bottom of the screen, rather than off the left edge).

Android had an animation framework back in the beginning, one that is still available for you today. However, Android 3.0 introduced a new *animator* framework that is going to be Android's primary focus for animated effects going forward. Many, but not all, of the animator framework capabilities are available to us as developers via a backport.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Also, you should read the chapter on custom views, to be able to make sense of one of the samples.

ViewPropertyAnimator

Let's say that you want to fade out a widget, instead of simply setting its visibility to `INVISIBLE` or `GONE`.

For a widget whose name is v, on API Level 11 or higher, that is as simple as:

```
v.animate().alpha(0);
```

Here, "alpha" refers to the "alpha channel". An alpha of 1 is normal opacity, while an alpha of 0 is completely transparent, with values in between representing various levels of translucence.

That may seem rather simple. The good news is, it *really* is that easy. Of course, there is a lot more you can do here, and you might have to worry about supporting older Android versions, and we need to think about things other than fading widgets in and out, and so forth.

First, though, let's consider what is really going on when we call `animate()` on a widget on API Level 11+.

Native Implementation

The call to `animate()` returns an instance of `ViewPropertyAnimator`. This object allows us to build up a description of an animation to be performed, such as calling `alpha()` to change the alpha channel value. `ViewPropertyAnimator` uses a so-called fluent interface, much like the various builder classes (e.g., `Notification.Builder`) — calling a method on a `ViewPropertyAnimator()` usually returns the `ViewPropertyAnimator` itself. This allows you to build up an animation via a chained series of method calls, starting with that call to `animate()` on the widget.

You will note that we do not end the chain of method calls with something like a `start()` method. `ViewPropertyAnimator` will automatically arrange to start the animation once we return control of the main application thread back to the framework. Hence, we do not have to explicitly start the animation.

You will also notice that we did not indicate any particulars about how the animation should be accomplished, beyond stating the ending alpha channel value of 0. `ViewPropertyAnimator` will use some standard defaults for the animation, such as a default duration, to determine how quickly Android changes the alpha value from its starting point to 0. Most of those particulars can be overridden from their defaults via additional methods called on our `ViewPropertyAnimator`, such as `setDuration()` to provide a duration in milliseconds.

There are four standard animations that `ViewPropertyAnimator` can perform:

1. Changes in alpha channel values, for fading widgets in and out
2. Changes in widget position, by altering the X and Y values of the upper-left corner of the widget, from wherever on the screen it used to be to some new value
3. Changes in the widget's rotation, around any of the three axes
4. Changes in the widget's size, where Android can scale the widget by some percentage to expand or shrink it

We will see an example of changing a widget's position, using the `translationXBy()` method, later in this chapter.

You are welcome to use more than one animation effect simultaneously, such as using both `alpha()` and `translationXBy()` to slide a widget horizontally and have it fade in or out.

There are other aspects of the animation that you can control. By default, the animation happens linearly — if we are sliding 500 pixels in 500ms, the widget will move evenly at 1 pixel/ms. However, you can specify a different "interpolator" to override that default linear behavior (e.g., start slow and accelerate as the animation proceeds). You can attach a listener object to find out about when the animation starts and ends. And, you can specify `withLayer()` to indicate that Android should try to more aggressively use hardware acceleration for an animation, a concept that we will get into in greater detail later in this chapter.

To see this in action, take a look at the `Animation/AnimatorFade` sample app.

The app consists of a single activity (`MainActivity`). It uses a layout that is dominated by a single `TextView` widget, whose ID is `fadee`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/fadee"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:text="@string/fading_out"
    android:textAppearance="?android:attr/textAppearanceLarge"
    tools:context=".MainActivity"/>
```

```
</RelativeLayout>
```

(from Animation/AnimatorFade/app/src/main/res/layout/activity_main.xml)

In `onCreate()`, we load up the layout and get our hands on the `fadee` widget:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  fadee=(TextView)findViewById(R.id.fadee);
}
```

(from Animation/AnimatorFade/app/src/main/java/com/commonsware/android/animator/fade/MainActivity.java)

`MainActivity` itself implements `Runnable`, and our `run()` method will perform some animated effects:

```
@Override
public void run() {
  if (fadingOut) {
    fadee.animate().alpha(0).setDuration(PERIOD);
    fadee.setText(R.string.fading_out);
  }
  else {
    fadee.animate().alpha(1).setDuration(PERIOD);
    fadee.setText(R.string.coming_back);
  }

  fadingOut=!fadingOut;

  fadee.postDelayed(this, PERIOD);
}
```

(from Animation/AnimatorFade/app/src/main/java/com/commonsware/android/animator/fade/MainActivity.java)

Specifically, we use `ViewPropertyAnimator` to fade out the `TextView` over a certain period (`fadee.animate().alpha(0).setDuration(PERIOD);`) and set its caption to a value indicating that we are fading out. If we are to be fading back in, we perform the opposite animation and set the caption to a different value. We then flip the `fadingOut` `boolean` for the next pass and use `postDelayed()` to reschedule ourselves to run after the period has elapsed.

To complete the process, we `run()` our code initially in `onStop()` and cancel the `postDelayed()` loop in `onStart()`:

```
@Override
public void onStart() {
  super.onStart();

  run();
}

@Override
public void onStop() {
  fadee.removeCallbacks(this);

  super.onStop();
}
```

(from Animation/AnimatorFade/app/src/main/java/com/commonsware/android/animator/fade/MainActivity.java)

The result is that the `TextView` smoothly fades out and in, alternating captions as it goes.

However, it would be really unpleasant if all this animator goodness worked only on API Level 11+. Fortunately for us, somebody wrote a backport.

Backport Via NineOldAndroids

Jake Wharton wrote NineOldAndroids. This is, in effect, a backport of `ViewPropertyAnimator` and its underpinnings. There are some slight changes in how you use it, because NineOldAndroids is simply a library. It cannot add methods to existing classes (like adding `animate()` to `View`), nor can it add capabilities that the underlying firmware simply lacks. But, it may cover many of your animator needs, even if the name is somewhat inexplicable, and it works going all the way back to API Level 1, ensuring that it will cover any Android release that you care about.

NineOldAndroids is an Android library project. Android Studio users can add a `implementation` statement to their `dependencies` closure in `build.gradle` to pull in `com.nineoldandroids:library:...` (for some version indicated by `...`).

Since NineOldAndroids cannot add `animate()` to `View`, the recommended approach is to use a somewhat obscure feature of Java: imported static methods. An `import static` statement, referencing a particular static method of a class, makes that method available as if it were a static method on the class that you are writing, or as

some sort of global function. NineOldAndroids has an `animate()` method that you can import this way, so instead of `v.animate()`, you use `animate(v)` to accomplish the same end. Everything else is the same, except perhaps some imports, to reference NineOldAndroids instead of the native classes.

You can see this in the `Animation/AnimatorFadeBC` sample app.

In addition to having the NineOldAndroids JAR in `libs/`, the only difference between this edition and the previous sample is in how the animation is set up. Instead of lines like:

```
fadee.animate().alpha(0).setDuration(PERIOD);
```

we have:

```
animate(fadee).alpha(0).setDuration(PERIOD);
```

This takes advantage of our static import:

```
import static com.nineoldandroids.view.ViewPropertyAnimator.animate;
```

If the static import makes you queasy, you are welcome to simply import the `com.nineoldandroids.view.ViewPropertyAnimator` class, rather than the static method, and call the `animate()` method on `ViewPropertyAnimator`:

```
ViewPropertyAnimator.animate(fadee).alpha(0).setDuration(PERIOD);
```

The Foundation: Value and Object Animators

`ViewPropertyAnimator` itself is a layer atop of a more primitive set of animators, known as value and object animators.

A `ValueAnimator` handles the core logic of transitioning some value, from an old to a new value, over a period of time. `ValueAnimator` offers replaceable "interpolators", which will determine how the values change from start to finish over the animation period (e.g., start slowly, accelerate, then end slowly). `ValueAnimator` also handles the concept of a "repeat mode", to indicate if the animation should simply happen once, a fixed number of times, or should infinitely repeat (and, in the latter cases, whether it does so always transitioning from start to finish or if it reverses direction on alternate passes, going from finish back to start).

What `ValueAnimator` does *not* do is actually change anything. It is merely computing the different values based on time. You can call `getAnimatedValue()` to find out the value at any point in time, or you can call `addUpdateListener()` to register a listener object that will be notified of each change in the value, so that change can be applied somewhere.

Hence, what tends to be a bit more popular is `ObjectAnimator`, a subclass of `ValueAnimator` that automatically applies the new values. `ObjectAnimator` does this by calling a setter method on some object, where you supply the object and the "property name" used to derive the getter and setter method names. For example, if you request a property name of `foo`, `ObjectAnimator` will try to call `getFoo()` and `setFoo()` methods on your supplied object.

As with `ViewPropertyAnimator`, `ValueAnimator` and `ObjectAnimator` are implemented natively in API Level 11 and are available via the NineOldAndroids backport as well.

To see what `ObjectAnimator` looks like in practice, let us examine the `Animation/ObjectAnimator` sample app.

Once again, our activity's layout is pretty much just a centered `TextView`, here named `word`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/word"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:textAppearance="?android:attr/textAppearanceLarge"
    tools:context=".MainActivity"/>

</RelativeLayout>
```

(from Animation/ObjectAnimator/app/src/main/res/layout/activity_main.xml)

The objective of our activity is to iterate through 25 words, showing one at a time in the `TextView`:

```
package com.commonsware.android.animator.obj;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import com.nineoldandroids.animation.ObjectAnimator;
import com.nineoldandroids.animation.ValueAnimator;

public class MainActivity extends Activity {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private TextView word=null;
  int position=0;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    word=(TextView)findViewById(R.id.word);

    ValueAnimator positionAnim = ObjectAnimator.ofInt(this, "wordPosition", 0, 24);
    positionAnim.setDuration(12500);
    positionAnim.setRepeatCount(ValueAnimator.INFINITE);
    positionAnim.setRepeatMode(ValueAnimator.RESTART);
    positionAnim.start();
  }

  public void setWordPosition(int position) {
    this.position=position;
    word.setText(items[position]);
  }

  public int getWordPosition() {
    return(position);
  }
}
```

(from Animation/ObjectAnimator/app/src/main/java/com/commonsware/android/animator/obj/MainActivity.java)

To accomplish this, we use NineOldAndroids version of `ObjectAnimator`, saying that we wish to "animate" the `wordPosition` property of the activity itself, from 0 to 24. We configure the animation to run for 12.5 seconds (i.e., 500ms per word) and to repeat indefinitely by restarting the animation from the beginning on each pass. We then call `start()` to kick off the animation.

For this to work, though, we need `getWordPosition()` and `setWordPosition()` accessor methods for the theoretical `wordPosition` property. In our case, the "word position" is simply an integer data member of the activity, which we return in `getWordPosition()` and update in `setWordPosition()`. However, we also update the `TextView` in `setWordPosition()`, to display the word at that position.

The net effect is that words appear in our `TextView`, changing on average every 500ms.

Animating Custom Types

In the previous section, we animated an `int` property of an `Activity`. That works, because Android knows how to compute `int` values between the start and end position, through simple math.

But, what if we wanted to animate something that is not a simple number? For example, what if we want to animate a `Color`, or a `LatLng` from Maps V2, or a `TastyTreat` class of our own design?

So long as *we* can perform the calculations, we can animate a type of anything we want, using `TypeEvaluator` and `ofObject()` on `ObjectAnimator`.

A `TypeEvaluator` is a simple interface, containing a single method that we need to override: `evaluate()`. However, `TypeEvaluator` uses generics, and so our implementation will actually be of some concrete class (e.g., a `TypeEvaluator` of `TastyTreat`). Our job in `evaluate()` is to return a value of our designated type (e.g., `TastyTreat`) given three inputs:

1. The initial value for our animation range, in the form of our designated type
2. The end value for our animation range, in the form of our designated type
3. The fraction along that range that represents how much we have moved from the initial value to the end value

Note that the fraction is not limited to being between 0 and 1, as certain interpolators (e.g., an overshoot interpolator) might result in a fraction being negative (e.g., we overshot past the initial value) or greater than one (e.g., we overshot past the end value).

For example, to have a `TypeEvaluator` of `Color`, we might have `evaluate()` generate a new `Color` instance based upon applying the fraction to the initial and end red, green, blue, and alpha channels.

To use a `TypeEvaluator`, instead of `ofInt()`, `ofFloat()`, or similar simple factory methods on `ObjectAnimator`, we use `ofObject()`. `ofObject()` takes the object to be animated, the property to be animated, the `TypeEvaluator` to assist in the actual

animation, and the final value of the animation (or, optionally, a series of waypoints to be animated along).

A flavor of `ofObject()` that takes the property name — akin to the `wordPosition ofInt()` used in the previous section — has been around since API Level 11. API Level 14 added an `ofObject()` method that takes a `Property` value instead of the name of the property. This version has the added benefit of type-safety, as it can ensure that your object to be animated, `TypeEvaluator`, and final position are all of the same type.

You can see an example of using `TypeEvaluator` this way in the chapter on Maps V2, as we animate the movement of a map marker from a starting point to an ending point.

Hardware Acceleration

Animated effects operate much more smoothly with hardware acceleration. There are two facets to employing hardware acceleration for animations: enabling it overall and directing its use for the animations themselves.

Hardware acceleration is enabled overall on Android devices running Android 4.0 or higher (API Level 14). On Android 3.x, hardware acceleration is available but is disabled by default — use `android:hardwareAccelerated="true"` in your `<application>` or `<activity>` element in the manifest to enable it on those versions. Hardware acceleration for 2D graphics operations like widget animations is not available on older versions of Android.

While this will provide some benefit across the board, you may also wish to consider rendering animated widgets or containers in an off-screen buffer, or "hardware layer", that then gets applied to the screen via the GPU. In particular, the GPU can apply certain animated transformations to a hardware layer without forcing software to redraw the widgets or containers (e.g., what happens when you `invalidate()` them). As it turns out, these GPU-enhanced transformations match the ones supported by `ViewPropertyAnimator`:

1. Changes in alpha channel values, for fading widgets in and out
2. Changes in widget position, by altering the X and Y values of the upper-left corner of the widget, from wherever on the screen it used to be to some new value
3. Changes in the widget's rotation, around any of the three axes

4. Changes in the widget's size, where Android can scale the widget by some percentage to expand or shrink it

By having the widget be rendered in a hardware layer, these `ViewPropertyAnimator` operations are significantly more efficient than before.

However, since hardware layers take up video memory, generally you do not want to keep a widget or container in a hardware layer indefinitely. Instead, the recommended approach is to have the widget or container be rendered in a hardware layer only while the animation is ongoing, by calling `setLayerType()` for `LAYER_TYPE_HARDWARE` before the animation begins, then calling `setLayerType()` for `LAYER_TYPE_NONE` (i.e., return to default behavior) when the animation completes. Or, for `ViewPropertyAnimator` on API Level 16 and higher, use `withLayer()` in the fluent interface to have it apply the hardware layer automatically just for the animation duration.

We will see examples of using hardware acceleration this way in the next section.

The Three-Fragment Problem

The original tablet implementation of Gmail organized its landscape main activity into two panes, one on the left taking up ~30% of the screen, and one on the right taking up the remainder:

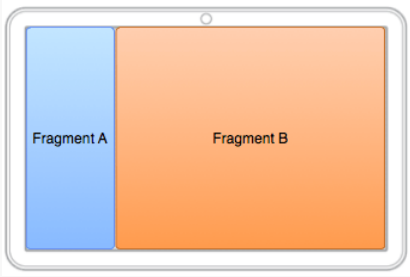


Figure 565: Gmail Fragments (image courtesy of Google and AOSP)

Gmail had a very specific navigation mode in its main activity when viewed in landscape on a tablet, where upon some UI event (e.g., tapping on something in the right-hand area):

- The original left-hand fragment (Fragment A) slid off the screen to the left

- The original right-hand fragment (Fragment B) slid to the left edge of the screen and shrunk to take up the spot vacated by Fragment A
- Another fragment (Fragment C) slid in from the right side of the screen and expanded to take up the spot vacated by Fragment B

And a BACK button press reversed this operation.

This is a bit tricky to set up, leading to the author of this book posting a question on Stack Overflow to get input. Here, we will examine one of the results of that discussion, based in large part on the implementation of the AOSP Email app, which has a similar navigation flow. The other answers on that question may have merit in other scenarios as well.

You can see one approach for implementing the three-pane solution in the `Animation/ThreePane` sample app.

The ThreePaneLayout

The logic to handle the animated effects is encapsulated in a `ThreePaneLayout` class. It is designed to be used in a layout XML resource where you supply the contents of the three panes, sizing the first two as you want, with the third "pane" having zero width at the outset:

```
<com.commonsware.android.anim.threepane.ThreePaneLayout xmlns:android="http://schemas.android.com/apk/res/
android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/root"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <FrameLayout
    android:id="@+id/left"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="3"/>

  <FrameLayout
    android:id="@+id/middle"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="7"/>

  <Button
    android:layout_width="0dp"
    android:layout_height="match_parent"/>

</com.commonsware.android.anim.threepane.ThreePaneLayout>
```

(from Animation/ThreePane/app/src/main/res/layout/activity_main.xml)

`ThreePaneLayout` itself is a subclass of `LinearLayout`, set up to always be horizontal, regardless of what might be set in the layout XML resource.

```
public ThreePaneLayout(Context context, AttributeSet attrs) {
  super(context, attrs);
  initSelf();
}

void initSelf() {
  setOrientation(HORIZONTAL);
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

When the layout finishes inflating, we grab the three panes (defined as the first three children of the container) and stash them in data members named `left`, `middle`, and `right`, with matching getter methods:

```
@Override
public void onFinishInflate() {
  super.onFinishInflate();

  left=getChildAt(0);
  middle=getChildAt(1);
  right=getChildAt(2);
}

public View getLeftView() {
  return(left);
}

public View getMiddleView() {
  return(middle);
}

public View getRightView() {
  return(right);
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

The major operational API, from the standpoint of an activity using `ThreePaneLayout`, is `hideLeft()` and `showLeft()`. `hideLeft()` will switch from showing the `left` and `middle` widgets in their original size and position to showing the `middle` and `right` widgets wherever `left` and `middle` had been originally. `showLeft()` reverses the operation.

The problem is that, initially, we do not know where the widgets are or how big they are, as that should be able to be set from the layout XML resource and are not known until the `ThreePaneLayout` is actually applied to the screen. Hence, we lazy-retrieve those values in `hideLeft()`, plus remove any weights that had been originally defined, setting the actual pixel widths on the widgets instead:

```
public void hideLeft() {
  if (leftWidth == -1) {
    leftWidth=left.getWidth();
    middleWidthNormal=middle.getWidth();
    resetWidget(left, leftWidth);
    resetWidget(middle, middleWidthNormal);
    resetWidget(right, middleWidthNormal);
    requestLayout();
  }

  translateWidgets(-1 * leftWidth, left, middle, right);

  ObjectAnimator.ofInt(this, "middleWidth", middleWidthNormal,
                       leftWidth).setDuration(ANIM_DURATION).start();
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

The work to change the weights into widths is handled in `resetWidget()`:

```
private void resetWidget(View v, int width) {
  LinearLayout.LayoutParams p=
      (LinearLayout.LayoutParams)v.getLayoutParams();

  p.width=width;
  p.weight=0;
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

After the lazy-initialization and widget cleanup, we perform the two animations. `translateWidgets()` will slide each of our three widgets to the left by the width of the `left` widget, using a `ViewPropertyAnimator` and a hardware layer:

```
private void translateWidgets(int deltaX, View... views) {
  for (final View v : views) {
    v.setLayerType(View.LAYER_TYPE_HARDWARE, null);

    v.animate().translationXBy(deltaX).setDuration(ANIM_DURATION)
     .setListener(new AnimatorListenerAdapter() {
```

```
        @Override
        public void onAnimationEnd(Animator animation) {
          v.setLayerType(View.LAYER_TYPE_NONE, null);
        }
      });
  }
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

The resize animation — to set the `middle` size to be what `left` had been – is handled via an `ObjectAnimator`, for a theoretical property of `middleWidth` on `ThreePaneLayout`. That is backed by a `setMiddleWidth()` method that adjusts the `width` property of the `middle` widget's `LayoutParams` and triggers a redraw:

```
@SuppressWarnings("unused")
private void setMiddleWidth(int value) {
  middle.getLayoutParams().width=value;
  requestLayout();
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

The `showLeft()` method simply performs those two animations in reverse:

```
public void showLeft() {
  translateWidgets(leftWidth, left, middle, right);

  ObjectAnimator.ofInt(this, "middleWidth", leftWidth,
                       middleWidthNormal).setDuration(ANIM_DURATION)
                .start();
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

Using the ThreePaneLayout

The sample app uses one activity (`MainActivity`) and one fragment (`SimpleListFragment`) to set up and use the `ThreePaneLayout`. The objective is a UI that roughly mirrors that of the AOSP Email app: a list on the left, a list in the middle (whose contents are based on the item chosen in the left list), and something else on the right (whose contents are based on the item chosen in the middle list).

SimpleListFragment is used for both lists. Its newInstance() factory method is handed the list of strings to display. SimpleListFragment just loads those into its ListView, also setting up CHOICE_MODE_SINGLE for use with the activated style, and routing all clicks on the list to the MainActivity that hosts the fragment:

```
package com.commonsware.android.anim.threepane;

import android.app.ListFragment;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import java.util.ArrayList;
import java.util.Arrays;

public class SimpleListFragment extends ListFragment {
  private static final String KEY_CONTENTS="contents";

  public static SimpleListFragment newInstance(String[] contents) {
    return(newInstance(new ArrayList<String>(Arrays.asList(contents))));
  }

  public static SimpleListFragment newInstance(ArrayList<String> contents) {
    SimpleListFragment result=new SimpleListFragment();
    Bundle args=new Bundle();

    args.putStringArrayList(KEY_CONTENTS, contents);
    result.setArguments(args);

    return(result);
  }

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    setContents(getArguments().getStringArrayList(KEY_CONTENTS));
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    ((MainActivity)getActivity()).onListItemClick(this, position);
  }

  void setContents(ArrayList<String> contents) {
    setListAdapter(new ArrayAdapter<String>(
```

```
                                        getActivity(),
                                        R.layout.simple_list_item_1,
                                        contents));
  }
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/SimpleListFragment.java)

`MainActivity` populates the `left` `FrameLayout` with a `SimpleListFragment` in `onCreate()`, if the fragment does not already exist (e.g., from a configuration change). When an item in the left list is clicked, `MainActivity` populates the `middle` `FrameLayout`. When an item in the middle list is clicked, it sets the caption of the `right` `Button` and uses `hideLeft()` to animate that `Button` onto the screen, hiding the `left` list. If the user presses BACK, and our `left` list is not showing, `MainActivity` calls `showLeft()` to reverse the animation:

```
package com.commonsware.android.anim.threepane;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import java.util.ArrayList;

public class MainActivity extends Activity {
  private static final String KEY_MIDDLE_CONTENTS="middleContents";
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private boolean isLeftShowing=true;
  private SimpleListFragment middleFragment=null;
  private ArrayList<String> middleContents=null;
  private ThreePaneLayout root=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    root=(ThreePaneLayout)findViewById(R.id.root);

    if (getFragmentManager().findFragmentById(R.id.left) == null) {
      getFragmentManager().beginTransaction()
                          .add(R.id.left,
                               SimpleListFragment.newInstance(items))
                          .commit();
```

```
  }

  middleFragment=
      (SimpleListFragment)getFragmentManager().findFragmentById(R.id.middle);
}

@Override
public void onBackPressed() {
  if (!isLeftShowing) {
    root.showLeft();
    isLeftShowing=true;
  }
  else {
    super.onBackPressed();
  }
}

@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putStringArrayList(KEY_MIDDLE_CONTENTS, middleContents);
}

@Override
protected void onRestoreInstanceState(Bundle inState) {
  middleContents=inState.getStringArrayList(KEY_MIDDLE_CONTENTS);
}

void onListItemClick(SimpleListFragment fragment, int position) {
  if (fragment == middleFragment) {
    ((Button)root.getRightView()).setText(middleContents.get(position));

    if (isLeftShowing) {
      root.hideLeft();
      isLeftShowing=false;
    }
  }
  else {
    middleContents=new ArrayList<String>();

    for (int i=0; i < 20; i++) {
      middleContents.add(items[position] + " #" + i);
    }

    if (getFragmentManager().findFragmentById(R.id.middle) == null) {
      middleFragment=SimpleListFragment.newInstance(middleContents);
      getFragmentManager().beginTransaction()
```

```
                            .add(R.id.middle, middleFragment).commit();
      }
      else {
        middleFragment.setContents(middleContents);
      }
    }
  }
}
```

(from Animation/ThreePane/app/src/main/java/com/commonsware/android/anim/threepane/MainActivity.java)

The Results

If you run this app on a landscape tablet running API Level 11 or higher, you start off with a single list of words on the left:

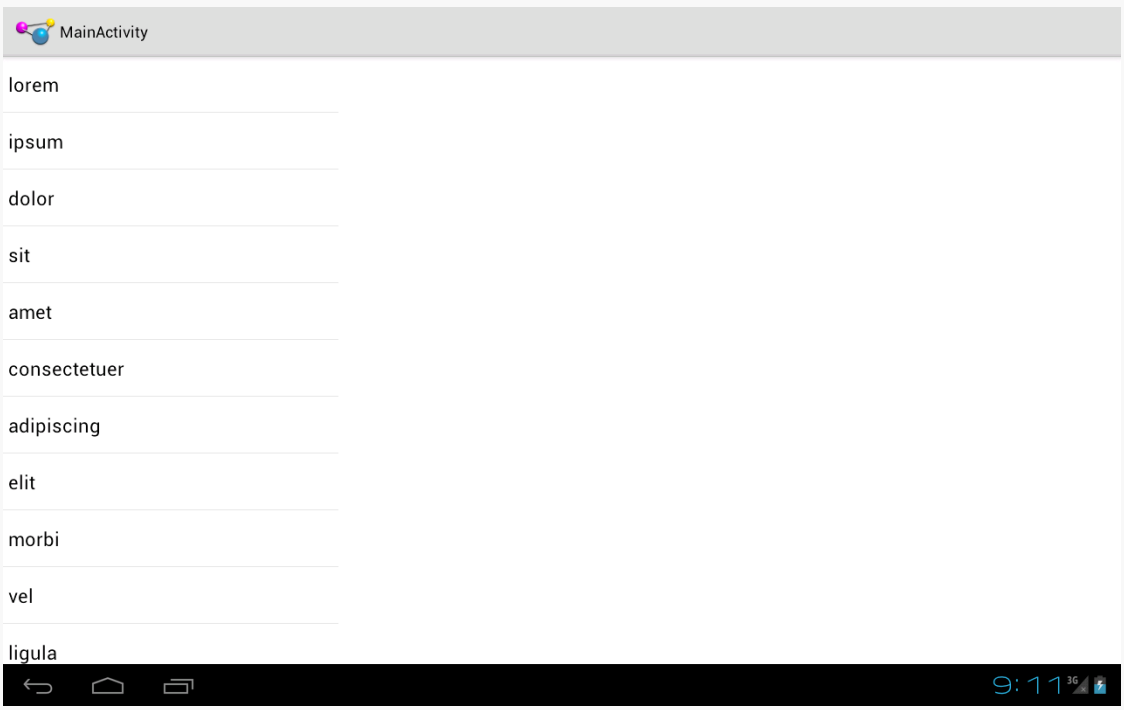


Figure 566: ThreePane, As Initially Launched

Clicking on a word brings up a second list, taking up the rest of the screen, with numbered entries based upon the clicked-upon word:

Figure 567: ThreePane, After Clicking a Word

Clicking on an entry in the second list starts the animation, sliding the first list off to the left, sliding the second list into the space vacated by the first list, and sliding in a "detail view" into the right portion of the screen:

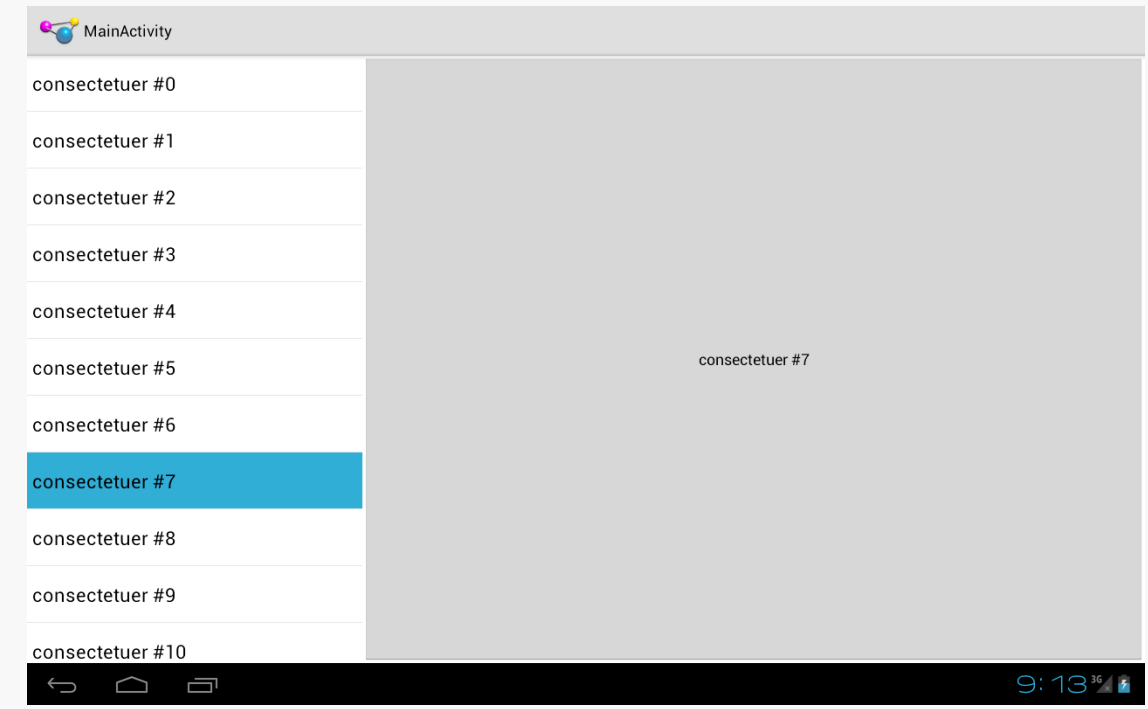


Figure 568: ThreePane, After Clicking a Numbered Word

Pressing BACK once will reverse the animation, restoring you to the two-list perspective.

The Backport

The `ThreePane` sample described above uses the native API Level 11 version of the animator framework and the native implementation of fragments. However, the same approach can work using the Android Support package's version of fragments and NineOldAndroids. You can see this in the `Animation/ThreePaneBC` sample app.

Besides changing the import statements and adding the NineOldAndroids JAR file, the only other changes of substance were:

- Using `ViewPropertyAnimator.animate(v)` instead of `v.animate()` in `translateWidgets()`
- Conditionally setting the hardware acceleration layers via `setLayerType()` in `translateWidgets()` based upon API level, as that method was only added in API Level 11

The smoothness of animations, though, will vary by hardware capabilities. For example, on a first-generation Kindle Fire, running Android 2.3, the backport works but is not especially smooth, while the animations are very smooth on more modern hardware where hardware acceleration can be applied.

The Problems

As we will see in the chapter on "jank", there is some stutter in the rendering of this app. Fixing it requires removing the animated change in the width of the middle pane, which in turn makes the animation itself look worse. More details on the analysis can be found in the "jank" chapter.

Legacy Animations

Before `ViewPropertyAnimator` and the rest of the animator framework were added in API Level 11, we had the original `Animation` base class and specialized animations based upon it, like `TranslateAnimation` for movement and `AlphaAnimation` for fades. On the whole, you will want to try to use the animator framework where possible, as the new system is more powerful and efficient than the legacy `Animation` approach. However, particularly for apps where the NineOldAndroids backport is insufficient, you may wish to use the legacy framework.

After an overview of the role of the animation framework, we go in-depth to animate the movement of a widget across the screen. We then look at alpha animations, for fading widgets in and out. We then see how you can get control during the lifecycle of an animation, how to control the acceleration of animations, and how to group animations together for parallel execution. Finally, we see how the same framework can now be used to control the animation for the switching of activities.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic resources and basic widgets. Also, you should read the chapter on custom views.

It's Not Just For Toons Anymore

Android has a package of classes (`android.view.animation`) dedicated to animating the movement and behavior of widgets.

They center around an `Animation` base class that describes what is to be done. Built-in animations exist to move a widget (`TranslateAnimation`), change the transparency of a widget (`AlphaAnimation`), revolve a widget (`RotateAnimation`), and resize a widget (`ScaleAnimation`). There is even a way to aggregate animations together into a composite `Animation` called an `AnimationSet`. Later sections in this chapter will examine the use of several of these animations.

Given that you have an animation, to apply it, you have two main options:

1. You may be using a container that supports animating its contents, such as a `ViewFlipper` or `TextSwitcher`. These are typically subclasses of `ViewAnimator` and let you define the "in" and "out" animations to apply. For example, with a `ViewFlipper`, you can specify how it flips between `Views` in terms of what animation is used to animate "out" the currently-visible `View` and what animation is used to animate "in" the replacement `View`.
2. You can simply tell any `View` to `startAnimation()`, given the `Animation` to apply to itself. This is the technique we will be seeing used in the examples in this chapter.

A Quirky Translation

Animation takes some getting used to. Frequently, it takes a fair bit of experimentation to get it all working as you wish. This is particularly true of `TranslateAnimation`, as not everything about it is intuitive, even to authors of Android books.

Mechanics of Translation

The simple constructor for `TranslateAnimation` takes four parameters describing how the widget should move: the before and after X offsets from the current position, and the before and after Y offsets from the current position. The Android documentation refers to these as `fromXDelta`, `toXDelta`, `fromYDelta`, and `toYDelta`.

In Android's pixel-space, an `(X,Y)` coordinate of `(0,0)` represents the upper-left corner of the screen. Hence, if `toXDelta` is greater than `fromXDelta`, the widget will move to the right, if `toYDelta` is greater than `fromYDelta`, the widget will move down, and so on.

Imagining a Sliding Panel

Some Android applications employ a sliding panel, one that is off-screen most of the time but can be called up by the user (e.g., via a menu) when desired. When anchored at the bottom of the screen, you get a container that slides up from the bottom and slides down and out when being removed.

One way to implement such a panel is to have a container (e.g., a `LinearLayout`) whose contents are absent (`INVISIBLE`) when the panel is closed and is present (`VISIBLE`) when the drawer is open. If we simply toggled `setVisibility()` using the aforementioned values, though, the panel would wink open and closed immediately, without any sort of animation. So, instead, we want to:

1. Make the panel visible and animate it up from the bottom of the screen when we open the panel
2. Animate it down to the bottom of the screen and make the panel invisible when we close the panel

The Aftermath

This brings up a key point with respect to `TranslateAnimation`: the animation temporarily moves the widget, but if you want the widget to stay where it is when the animation is over, you have to handle that yourself. Otherwise, the widget will snap back to its original position when the animation completes.

In the case of the panel opening, we handle that via the transition from `INVISIBLE` to `VISIBLE`. Technically speaking, the panel is always "open", in that we are not, in the end, changing its position. But when the body of the panel is `INVISIBLE`, it takes up no space on the screen; when we make it `VISIBLE`, it takes up whatever space it is supposed to.

Later in this chapter, we will cover how to use animation listeners to accomplish this end for closing the panel.

Introducing SlidingPanel

With all that said, turn your attention to the `Animation/SlidingPanel` sample project and, in particular, the `SlidingPanel` class.

This class implements a layout that works as a panel, anchored to the bottom of the screen. A `toggle()` method can be called by the activity to hide or show the panel. The panel itself is a `LinearLayout`, so you can put whatever contents you want in there.

We use two flavors of `TranslateAnimation`, one for opening the panel and one for closing it.

Here is the opening animation:

```
    anim=new TranslateAnimation(0.0f, 0.0f,
                                getHeight(),
                                0.0f);
```

(from Animation/SlidingPanel/app/src/main/java/com/commonsware/android/anim/SlidingPanel.java)

Our `fromXDelta` and `toXDelta` are both `0`, since we are not shifting the panel's position along the horizontal axis. Our `fromYDelta` is the panel's height according to its layout parameters (representing how big we want the panel to be), because we want the panel to start the animation at the bottom of the screen; our `toYDelta` is `0` because we want the panel to be at its "natural" open position at the end of the animation.

Conversely, here is the closing animation:

```
    anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                getHeight());
```

(from Animation/SlidingPanel/app/src/main/java/com/commonsware/android/anim/SlidingPanel.java)

It has the same basic structure, except the Y values are reversed, since we want the panel to start open and animate to a closed position.

The result is a container that can be closed:

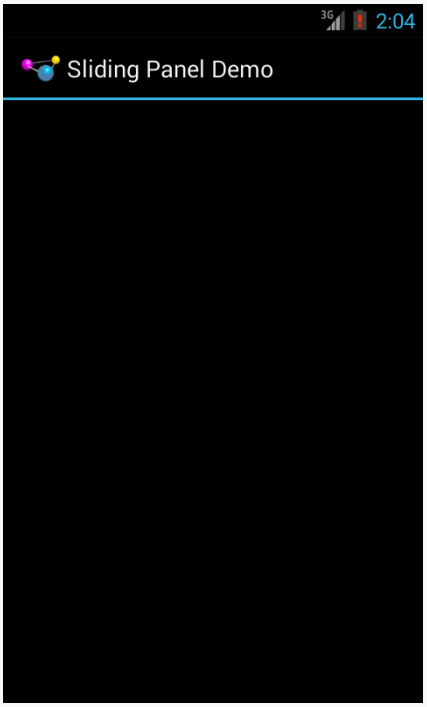


Figure 569: The SlidingPanel sample application, with the panel closed

... or open, in this case toggled via a menu choice in the `SlidingPanelDemo` activity:

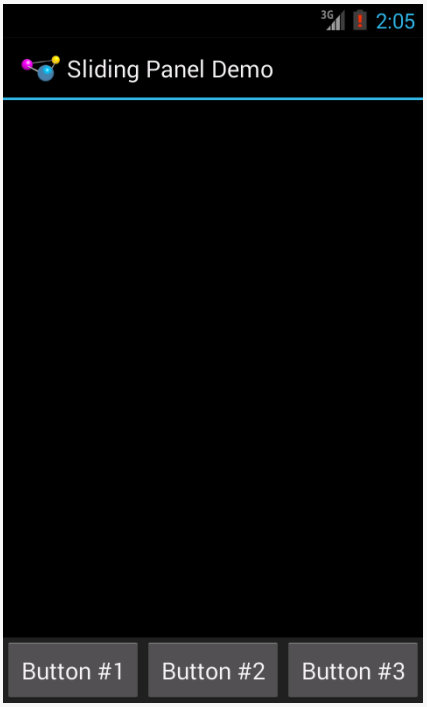


Figure 570: The SlidingPanel sample application, with the panel open

Using the Animation

When setting up an animation, you also need to indicate how long the animation should take. This is done by calling `setDuration()` on the animation, providing the desired length of time in milliseconds.

When we are ready with the animation, we simply call `startAnimation()` on the `SlidingPanel` itself, causing it to move as specified by the `TranslateAnimation` instance.

Fading To Black. Or Some Other Color.

`AlphaAnimation` allows you to fade a widget in or out by making it less or more transparent. The greater the transparency, the more the widget appears to be "fading".

Alpha Numbers

You may be used to alpha channels, when used in `#AARRGGBB` color notation, or perhaps when working with alpha-capable image formats like PNG.

Similarly, `AlphaAnimation` allows you to change the alpha channel for an entire widget, from fully-solid to fully-transparent.

In Android, a float value of `1.0` indicates a fully-solid widget, while a value of `0.0` indicates a fully-transparent widget. Values in between, of course, represent various amounts of transparency.

Hence, it is common for an `AlphaAnimation` to either start at `1.0` and smoothly change the alpha to `0.0` (a fade) or vice versa.

Animations in XML

With `TranslateAnimation`, we showed how to construct the animation in Java source code. One can also create animation resources, which define the animations using XML. This is similar to the process for defining layouts, albeit much simpler.

For example, there is a second animation project, `Animation/SlidingPanelEx`, which demonstrates a panel that fades out as it is closed. In there, you will find a `res/anim/` directory, which is where animation resources should reside. In there, you will find `fade.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromAlpha="1.0"
  android:toAlpha="0.0" />
```

(from Animation/SlidingPanelEx/app/src/main/res/anim/fade.xml)

The name of the root element indicates the type of animation (in this case, alpha for an `AlphaAnimation`). The attributes specify the characteristics of the animation, in this case a fade from `1.0` to `0.0` on the alpha channel.

This XML is the same as calling `new AlphaAnimation(1.0f,0.0f)` in Java.

Using XML Animations

To make use of XML-defined animations, you need to inflate them, much as you might inflate a `View` or `Menu` resource. This is accomplished by using the `loadAnimation()` static method on the `AnimationUtils` class, seen here in our `SlidingPanel` constructor:

```
public SlidingPanel(final Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  TypedArray a=ctxt.obtainStyledAttributes(attrs,
                                           R.styleable.SlidingPanel,
                                           0, 0);

  speed=a.getInt(R.styleable.SlidingPanel_speed, 300);

  a.recycle();

  fadeOut=AnimationUtils.loadAnimation(ctxt, R.anim.fade);
}
```

(from Animation/SlidingPanelEx/app/src/main/java/com/commonsware/android/anim2/SlidingPanel.java)

Here, we are loading our fade animation, given a `Context`. This is being put into an `Animation` variable, so we neither know nor care that this particular XML that we are loading defines an `AlphaAnimation` instead of, say, a `RotateAnimation`.

When It's All Said And Done

Sometimes, you need to take action when an animation completes.

For example, when we close the panel, we want to use a `TranslationAnimation` to slide it down from the open position to closed... then *keep* it closed. With the system used in `SlidingPanel`, keeping the panel closed is a matter of calling `setVisibility()` on the contents with `INVISIBLE`.

However, you cannot do that when the animation begins; otherwise, the panel is gone by the time you try to animate its motion.

Instead, you need to arrange to have it become invisible when the animation ends. To do that, you use an animation listener.

An animation listener is simply an instance of the `AnimationListener` interface, provided to an animation via `setAnimationListener()`. The listener will be invoked when the animation starts, ends, or repeats (the latter courtesy of `CycleInterpolator`, discussed later in this chapter). You can put logic in the `onAnimationEnd()` callback in the listener to take action when the animation finishes.

For example, here is the `AnimationListener` for `SlidingPanel`:

```
Animation.AnimationListener collapseListener=new Animation.AnimationListener() {
  public void onAnimationEnd(Animation animation) {
    setVisibility(View.INVISIBLE);
  }

  public void onAnimationRepeat(Animation animation) {
    // not needed
  }

  public void onAnimationStart(Animation animation) {
    // not needed
  }
};
```

(from Animation/SlidingPanel/app/src/main/java/com/commonsware/android/anim/SlidingPanel.java)

All we do is set our content's visibility to be `INVISIBLE`, thereby closing the panel.

Loose Fill

You will see attributes, available on `Animation`, named `android:fillEnabled` and `android:fillAfter`. Reading those, you may think that you can dispense with the `AnimationListener` and just use those to arrange to have your widget wind up being "permanently" in the state represented by the end of the animation. All you would have to do is set each of those to true in your animation XML (or the equivalent in Java), and you would be set.

At least for `TranslateAnimation`, you would be mistaken.

It actually will look like it works — the animated widgets will be drawn in their new location. However, if those widgets are clickable, they will not be clicked in their new location, but rather in their old one. This, of course, is not terribly useful.

Hence, even though it is annoying, you will want to use the `AnimationListener` techniques described in this chapter.

Hit The Accelerator

In addition to the `Animation` classes themselves, Android also provides a set of `Interpolator` classes. These provide instructions for how an animation is supposed to behave during its operating period.

For example, the `AccelerateInterpolator` indicates that, during the duration of an animation, the rate of change of the animation should begin slowly and accelerate until the end. When applied to a `TranslateAnimation`, for example, the sliding movement will start out slowly and pick up speed until the movement is complete.

There are several implementations of the `Interpolator` interface besides `AccelerateInterpolator`, including:

1. `AccelerateDecelerateInterpolator`, which starts slowly, picks up speed in the middle, and slows down again at the end
2. `DecelerateInterpolator`, which starts quickly and slows down towards the end
3. `LinearInterpolator`, the default, which indicates the animation should proceed smoothly from start to finish
4. `CycleInterpolator`, which repeats an animation for a number of cycles, following the `AccelerateDecelerateInterpolator` pattern (slow, then fast, then slow)

To apply an interpolator to an animation, simply call `setInterpolator()` on the animation with the `Interpolator` instance, such as the following line from `SlidingPanel`:

```
    anim.setInterpolator(new AccelerateInterpolator(1.0f));
```

(from Animation/SlidingPanel/app/src/main/java/com/commonsware/android/anim/SlidingPanel.java)

You can also specify one of the stock interpolators via the `android:interpolator` attribute in your animation XML file.

Animate. Set. Match.

For the `Animation/SlidingPanelEx` project, though, we want the panel to slide open, but also fade when it slides closed. This implies two animations working at the same time (a fade and a slide). Android supports this via the `AnimationSet` class.

An `AnimationSet` is itself an `Animation` implementation. Following the composite design pattern, it simply cascades the major `Animation` events to each of the animations in the set.

To create a set, just create an `AnimationSet` instance, add the animations, and configure the set. For example, here is the logic from the `SlidingPanel` implementation in `Animation/SlidingPanelEx`:

```
public void toggle() {
  TranslateAnimation anim=null;
  AnimationSet set=new AnimationSet(true);

  isOpen=!isOpen;

  if (isOpen) {
    setVisibility(View.VISIBLE);
    anim=new TranslateAnimation(0.0f, 0.0f,
                                getHeight(),
                                0.0f);
  }
  else {
    anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                getHeight());
    anim.setAnimationListener(collapseListener);
    set.addAnimation(fadeOut);
  }

  set.addAnimation(anim);
  set.setDuration(speed);
  set.setInterpolator(new AccelerateInterpolator(1.0f));
  startAnimation(set);
}
```

(from Animation/SlidingPanelEx/app/src/main/java/com/commonsware/android/anim2/SlidingPanel.java)

If the panel is to be opened, we make the contents visible (so we can animate the motion upwards), and create a `TranslateAnimation` for the upward movement. If the panel is to be closed, we create a `TranslateAnimation` for the downward movement, but also add a pre-defined `AlphaAnimation` (`fadeOut`) to an `AnimationSet`. In either case, we add the `TranslateAnimation` to the set, give the set a duration and interpolator, and run the animation.

Active Animations

Starting with Android 1.5, users could indicate if they wanted to have inter-activity animations: a slide-in/slide-out effect as they switched from activity to activity. However, at that time, they could merely toggle this setting on or off, and applications had no control over these animations whatsoever.

Starting in Android 2.0, though, developers have a bit more control. Specifically:

1. Developers can call `overridePendingTransition()` on an `Activity`, typically after calling `startActivity()` to launch another activity or `finish()` to close up the current activity. The `overridePendingTransition()` indicates an in/out animation pair that should be applied as control passes from this activity to the next one, whether that one is being started (`startActivity()`) or is the one previous on the stack (`finish()`).
2. Developers can start an activity via an `Intent` containing the `FLAG_ACTIVITY_NO_ANIMATION` flag. As the name suggests, this flag requests that animations on the transitions involving this activity be suppressed.

These are prioritized as follows:

- Any call to `overridePendingTransition()` is always taken into account
- Lacking that, `FLAG_ACTIVITY_NO_ANIMATION` will be taken into account
- In the normal case, where neither of the two are used, whatever the user's preference, via the Settings application, is applied

Custom Drawables

Many times, our artwork can simply be some PNG or JPEG files, perhaps with different variations in different resource directories by density.

Sometimes, though, we need something more.

In addition to supporting standard PNG and JPEG files, Android has a number of custom drawable resource formats — mostly written in XML — that handle specific scenarios.

For example, you may wish to customize "the background" of a `Button`, but a `Button` really has several different background images for different circumstances (normal, pressed, focused, disabled, etc.). Android has a certain type of drawable resource that aggregates other drawable resources, indicating which of those other resources should be used in different circumstances (e.g., for a normal button use X, for a disabled button use Y).

In this chapter, we will explore these non-traditional types of "drawables" and how you can use them within your apps.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic resources, basic widgets, and vector drawables.

Having read the chapters on animators and legacy animations would be useful.

Where Do These Things Go?

All of the drawables described in this chapter, unless otherwise noted, are density-independent. Hence, they do not normally go in a density-dependent directories like `res/drawable-hdpi/`. However, that still leaves three possible candidates: `res/drawable-nodpi/`, `res/drawable-anydpi/`, and the unadorned `res/drawable/`.

nodpi: Fallback

A drawable in `res/drawable-nodpi/` is valid for any screen density. However, if there is another drawable with the same base name in a density-specific directory, and the device running your app happens to have that screen density, the density-specific resource will be used. As a result, `-nodpi` becomes a fallback, to be used in cases where you do not have something specific for a density.

For example, suppose that we have `res/drawable-nodpi/foo.xml` and `res/drawable-xxhdpi/foo.png`. An `-xxhdpi` device would use the PNG; all other devices would use the XML.

anydpi: Takeover

A drawable in `res/drawable-anydpi/` also is valid for any screen density. However, in this case, the `-anydpi` variant trumps any density-specific variant.

For example, suppose that we have `res/drawable-anydpi/foo.xml` and `res/drawable-xxhdpi/foo.png`. **All devices** would use the XML, even `-xxhdpi` devices.

For this reason, often you will see `-anydpi` used in conjunction with other qualifiers. A popular one will be `-v21`, to restrict the resources to be used on API Level 21+ devices.

No Qualifier: Just Say "WTF?"

`res/drawable/` is a synonym for `res/drawable-mdpi/`, for backwards compatibility with *really* old Android apps, written before we had density-specific resources. Hence, `res/drawable/` is not really an appropriate choice for density-independent drawables.

Alas, Android Studio may put some drawables here, for uncertain reasons.

So long as there are no other resources with the same basename, the choice made by Android Studio's developers is unlikely to cause any harm.

ColorDrawable

The simplest XML drawable format, by far, is for `ColorDrawable`. Not surprisingly, this defines a `Drawable` that is a solid color.

So, you can have a `res/drawable/thing.xml` file, containing something like this:

```
<color xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="#80FF00FF"/>
```

From there, you can use `@drawable/thing` or `R.drawable.thing` in the same places that you would use any other drawable resource.

Note that a `ColorDrawable` is different than a color resource. A color resource (e.g., `res/values/colors.xml`) specifies a color. A `ColorDrawable` resource defines a `Drawable` of a color. A `ColorDrawable` resource is welcome to reference a color defined by a color resource, though:

```
<color xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="@color/primary_dark"/>
```

AnimationDrawable

The original way of doing animation on the Web was via the animated GIF. An individual GIF file could contain many frames, and the browser would switch between those frames to display a basic animated effect. This was used by Web designers for things both good (animated progress "spinners") and bad ("hit the monkey" ad banners).

Android, on the whole, does not support animated GIF files, certainly not as regular images for use with widgets like `ImageView`.

However, there are times where having this sort of frame-by-frame animation would be useful. For example, in another chapter, we will look at `ProgressBar`, which is Android's primary way of demonstrating progress of background work. You may wish to customize the "spinning wheel" image that Android uses by default, to match your app's color scheme, or to spin your company logo, or whatever. On the

Web, particularly on older browsers, you might use an animated GIF for that. On Android, you still could, though it would require a third-party library or some fairly heavyweight solutions (e.g., `WebView`, `Movie`).

Another possibility is to use an `AnimationDrawable`. `AnimationDrawable` has the net effect of an animated GIF:

- You define a series of images that serve as the frames of the animation
- You define how long each of those images should be on the screen
- You define whether the animation should loop back to the beginning after it reaches the end or not

However, rather than encoding all of this in an animated GIF, you instead encode this information in an XML file, stored as a drawable resource.

XML-encoded drawable resources are typically stored in a drawable directory that does *not* contain density information, such as `res/drawable/`. That is because the XML-encoded drawable resources are density-invariant: they behave the same regardless of density. Those, like the `AnimationDrawable`, that refer to other images might well refer to other images that are stored in density-dependent resource directories, but the XML-encoded drawable itself is independent of density.

An `AnimationDrawable` is defined as in XML with a root `<animation-list>` element, containing a series of `<item>` elements for each frame:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/frame1" android:duration="250" />
    <item android:drawable="@drawable/frame2" android:duration="250" />
    <item android:drawable="@drawable/frame3" android:duration="250" />
    <item android:drawable="@drawable/frame4" android:duration="250" />
</animation-list>
```

The root `<animation-list>` element can have an `android:oneshot` attribute, indicating whether the animation should repeat after displaying the last frame (`false`) or stop (`true`).

The `<item>` elements have `android:drawable` attributes pointing to the individual images for the individual frames. Usually these frames are PNG or JPEG files, but you refer to them as drawable resources, using `@drawable` syntax, so Android can find the right image based upon the density (or other characteristics) of the current device. The `<item>` elements also need an `android:duration` attribute, specifying

the time in milliseconds that this frame should be on the screen. While the above example has all durations the same, that is not required.

For example, the Android OS uses `AnimationDrawable` resources in a few places. One is for the download icon used in a `Notification` for use with `DownloadManager` and similar situations. That drawable resource – `stat_sys_download.xml` — looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/* //device/apps/common/res/drawable/status_icon_background.xml
**
** Copyright 2008, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->
<animation-list
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/stat_sys_download_anim0" android:duration="200"
/>
    <item android:drawable="@drawable/stat_sys_download_anim1" android:duration="200"
/>
    <item android:drawable="@drawable/stat_sys_download_anim2" android:duration="200"
/>
    <item android:drawable="@drawable/stat_sys_download_anim3" android:duration="200"
/>
    <item android:drawable="@drawable/stat_sys_download_anim4" android:duration="200"
/>
    <item android:drawable="@drawable/stat_sys_download_anim5" android:duration="200"
/>
</animation-list>
```

Here, we have a repeating animation (`android:oneshot="false"`), consisting of six frames, each on the screen for 200 milliseconds.

By specifying an `AnimationDrawable` in your `Notification` for its icon, you too can have this sort of animated effect. Of course, the animation is "fire and forget": other than by removing or replacing the `Notification`, you cannot affect the animation in any other way.

Animated GIF Conversion

It may be that you have an animated GIF that you would like to use as the basis for your `AnimationDrawable`. If you have passing familiarity with Ruby, the author of this book has published a Ruby script, named `gif2animdraw`, that automates the conversion.

To use `gif2animdraw`, in addition to the script itself and a Ruby interpreter, you will need the `RMagick`, `slop`, and `builder` gems. Note that `RMagick`, in turn, will require ImageMagick libraries and therefore is a bit more complicated to install than is your ordinary gem.

On Linux environments, you can also `chmod` the script to run it directly; otherwise, you would run it via the `ruby` command.

The script takes four command-line switches:

- `-i` should point to the GIF file to be converted
- `-o` should point to the root output directory, which typically would be a project's `res/` directory
- `-d` should have, as a value, one of the Android density bucket names (e.g., `hdpi`); this will be used as the density for the frames of the GIF
- Optionally, include `--oneshot` to indicate that this should be a one-shot animation, not a repeating one

The results will be:

- A `drawable/` directory underneath your supplied root, containing a file with the same name as the GIF file, but with a `.xml` extension, representing the `AnimationDrawable` itself
- A `drawable-XXXX/` directory, where `XXXX` is your stated density, containing each frame of the animated GIF, as a PNG file, with a sequentially numbered filename based on the GIF's filename

StateListDrawable

Another XML-defined drawable resource, the `StateListDrawable`, is key if you want to have different images when widgets are in different states.

As outlined in the introduction to this chapter, what makes a `Button` visually be a `Button` is its background. To handle different looks for the `Button` background for different states (normal, pressed, disabled, etc.), the standard `Button` background is a `StateListDrawable`, one that looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_window_focused="false" android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_window_focused="false" android:state_enabled="false"
        android:drawable="@drawable/btn_default_normal_disable" />
    <item android:state_pressed="true"
        android:drawable="@drawable/btn_default_pressed" />
    <item android:state_focused="true" android:state_enabled="true"
        android:drawable="@drawable/btn_default_selected" />
    <item android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_focused="true"
        android:drawable="@drawable/btn_default_normal_disable_focused" />
    <item
         android:drawable="@drawable/btn_default_normal_disable" />
</selector>
```

The XML has a `<selector>` root element, indicating this is a `StateListDrawable`. The `<item>` elements inside the root describe what Drawable resource should be

used if the StateListDrawable is being used in some state. For example, if the "window" (think activity or dialog) does not have the focus (android:state_window_focused="false") and the Button is enabled (android:state_enabled="true"), then we use the @drawable/btn_default_normal Drawable resource. That resource, as it turns out, is a nine-patch PNG file, described later in this chapter.

Android applies each rule in turn, top-down, to find the Drawable to use for a given state of the StateListDrawable. The last rule has no android:state_* attributes, meaning it is the overall default image to use if none of the other rules match.

So, if you want to change the background of a Button, you need to:

- Copy the above resource, found in your Android SDK as res/drawable/btn_default.xml inside any of the platforms/ directories, into your project
- Copy each of the Button state nine-patch images into your project
- Modify whichever of those nine-patch images you want, to affect the visual change you seek
- If need be, tweak the states and images defined in the StateListDrawable XML you copied
- Reference the local StateListDrawable as the background for your Button

The backgrounds of most widgets that have backgrounds by default will use a StateListDrawable. Searching a platform version's res/drawable/ directory for XML files containing <selector> elements comes up with a rather long list.

ColorStateList

A ColorStateList is analogous to a StateListDrawable, in that it defines states and identifies what should be used for a given state. Whereas StateListDrawable ties states to drawables, ColorStateList ties states to colors. This allows you to, say, change the color of some text based upon whether that text is drawn in a widget that is being pressed, or has the focus, or is disabled. If you tailor the background of a text-based widget using a StateListDrawable, you may well wind up tailoring the foreground text using a ColorStateList.

While this chapter mentions ColorStateList, technically a ColorStateList is not a Drawable. You do not use it in methods that take drawables or in widget XML attributes that take drawables. Rather, there are other methods and other attributes that take a ColorStateList, such as android:textColor.

Similarly, while you can define a `ColorStateList` in XML, you do not do so in a `res/drawable/` resource directory, but rather a `res/color/` resource directory. Beyond that, though, a `ColorStateList` XML resource looks a lot like a `StateListDrawable` XML resource, such as this definition of `@android:color/primary_text_dark` from Android 4.4:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_enabled="false" android:color="@android:color/
bright_foreground_dark_disabled"/>
    <item android:state_window_focused="false" android:color="@android:color/
bright_foreground_dark"/>
    <item android:state_pressed="true" android:color="@android:color/
bright_foreground_dark_inverse"/>
    <item android:state_selected="true" android:color="@android:color/
bright_foreground_dark_inverse"/>
    <item android:state_activated="true" android:color="@android:color/
bright_foreground_dark_inverse"/>
    <item android:color="@android:color/bright_foreground_dark"/> <!-- not selected
-->
</selector>
```

Based upon the state, the `ColorStateList` pulls in a separate resource to define the actual color. Those colors, in turn, are defined via `<color>` elements in `res/values/colors.xml` as color resources, or are pulled in from system-defined colors (`@android:color/...` syntax):

```
    <color name="background_dark">#ff000000</color>
    <color name="background_light">#ffffffff</color>
    <color name="bright_foreground_dark">@android:color/background_light</color>
    <color name="bright_foreground_light">@android:color/background_dark</color>
```

```
    <color name="bright_foreground_dark_disabled">#80ffffff</color>
    <color name="bright_foreground_light_disabled">#80000000</color>
    <color name="bright_foreground_dark_inverse">@android:color/
bright_foreground_light</color>
    <color name="bright_foreground_light_inverse">@android:color/
bright_foreground_dark</color>
```

LayerDrawable

A `LayerDrawable` basically stacks a bunch of other drawables on top of each other. Later drawables are drawn on top of earlier drawables, much as later children of a `RelativeLayout` are drawn on top of earlier children.

Typically, you will create a `LayerDrawable` via a `<layer-list>` XML drawable resource.

For example, a `ToggleButton` widget has a `LayerDrawable` as its background:

```
?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+android:id/background" android:drawable="@android:drawable/
btn_default_small" />
    <item android:id="@+android:id/toggle" android:drawable="@android:drawable/
btn_toggle" />
</layer-list>
```

This `LayerDrawable` draws two images on top of each other. One is a standard small button background (`@android:drawable/btn_default_small`). The other is the

actual face of the toggle itself — a `StateListDrawable` that uses different images for checked and unchecked states.

In the `<layer-list>`, you can have several `<item>` elements. Each `<item>` element usually will need an `android:drawable` attribute, pointing to the drawable that should be drawn. Optionally, you can assign ID values to the items via `android:id` attributes, much like you would do for widgets in a layout XML resource. Later on, you can call `findDrawableByLayerId()` on the `LayerDrawable` to retrieve an individual `Drawable` representing the layer, given its `android:id` value.

There are also `android:left`, `android:right`, `android:top`, and `android:bottom` attributes, which you can use to provide dimension values to offset an image within the layered set. For example, you could use `android:left` to inset one of the layers by a certain number of pixels (or `dp` or whatever).

By default, the layers in the `LayerDrawable` are scaled to fit the size of whatever `View` is holding them (e.g., the size of the `ToggleButton` using the `LayerDrawable` as a background). To prevent this, you can skip the `android:drawable` attribute, and instead nest a `<bitmap>` element inside the `<item>`, where you can provide an `android:gravity` attribute to control how the image should be handled relative to its containing `View`. We will get more into nested `<bitmap>` elements later in this chapter.

TransitionDrawable

A `TransitionDrawable` is a `LayerDrawable` with one added feature: for a two-layer drawable, it can smoothly transition from showing one layer to another on top.

For example, you may have noticed that when you tap-and-hold on a row in a `ListView` that the selector highlight has an animated effect, slowly shifting colors from the color used for a simple click to one signifying that you have long-clicked the row. Android accomplishes this via a `TransitionDrawable`, set up as a `<transition>` XML drawable resource:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at
```

```
          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@android:drawable/list_selector_background_pressed"  />
    <item android:drawable="@android:drawable/list_selector_background_longpress"  />
</transition>
```

The `TransitionDrawable` object has a `startTransition()` method that you can use, that will have Android smoothly switch from the first drawable to the second. You specify the duration of the transition as a number of milliseconds passed to `startTransition()`. There are also options to reverse the transition, set up more of a cross-fade effect, and the like.

LevelListDrawable

A `LevelListDrawable` is similar in some respects to a `StateListDrawable`, insofar as one specific item from the "list drawable" will be displayed based upon certain conditions. In the case of `StateListDrawable`, the conditions are based upon the state of the widget using the drawable (e.g., checked, pressed, disabled). In the case of `LevelListDrawable`, it is merely an integer level.

For example, the status or system bar of your average Android device has an icon indicating the battery charge level. That is actually implemented as a `LevelListDrawable`, via an XML resource containing a root `<level-list>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/* //device/apps/common/res/drawable/stat_sys_battery.xml
**
** Copyright 2007, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
```

```
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->

<level-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:maxLevel="4" android:drawable="@android:drawable/
stat_sys_battery_0" />
    <item android:maxLevel="15" android:drawable="@android:drawable/
stat_sys_battery_15" />
    <item android:maxLevel="35" android:drawable="@android:drawable/
stat_sys_battery_28" />
    <item android:maxLevel="49" android:drawable="@android:drawable/
stat_sys_battery_43" />
    <item android:maxLevel="60" android:drawable="@android:drawable/
stat_sys_battery_57" />
    <item android:maxLevel="75" android:drawable="@android:drawable/
stat_sys_battery_71" />
    <item android:maxLevel="90" android:drawable="@android:drawable/
stat_sys_battery_85" />
    <item android:maxLevel="100" android:drawable="@android:drawable/
stat_sys_battery_100" />
</level-list>
```

This `LevelListDrawable` has eight items, whose `android:drawable` attributes point to specific other drawable resources (in this case, standard PNG files with different implementations for different densities). Each `<item>` has an `android:maxLevel` value. When someone calls `setLevel()` on the `Drawable` or `setImageLevel()` on the `ImageView`, Android will choose the item with the lowest `maxLevel` that meets or exceeds the requested level, and show that. In the case of the battery icon, when the battery level changes, the status bar picks up that change and calls `setImageLevel()` with the battery charge percentage (expressed as an integer from 0-100) — that, in turn, triggers the right PNG file to be displayed.

Another use of `LevelListDrawable` is with a `RemoteViews`, such as for an app widget. The `setImageLevel()` method is "remotable", despite not being directly part of the `RemoteViews` API. Hence, given that you use a `LevelListDrawable` in your app widget's layout, you should be able to use `setInt()` with a method name of `"setImageLevel"` to have the app widget update to display the proper image.

ScaleDrawable and ClipDrawable

A `ScaleDrawable` does pretty much what its name suggests: it scales another drawable. A `ClipDrawable` does pretty much what *its* name suggests: it clips another drawable.

How they do this, and how you control it, requires a bit more explanation.

Like `LevelListDrawable`, `ScaleDrawable` and `ClipDrawable` leverage the `setLevel()` method on `Drawable` (or the `setImageLevel()` method on `ImageView`). Whereas `LevelListDrawable` uses this to choose an individual image out of a set of possible images, `ScaleDrawable` and `ClipDrawable` use the level to control how much an image should be scaled or clipped. For this, they support a range of levels from 0 to 10000.

Scaling

For a level of 0, `ScaleDrawable` will not draw anything. For a level from 1 to 10000, `ScaleDrawable` will scale an image from a configurable minimum size to the bounds of the `View` to which the drawable is applied.

The amount of scaling is determined by `android:scaleHeight` and `android:scaleWidth` attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android"
  android:drawable="@android:drawable/btn_default"
  android:scaleGravity="left|top"
  android:scaleHeight="50%"
  android:scaleWidth="50%"/>
```

(from Drawable/ScaleClip/app/src/main/res/drawable/scale.xml)

The above `ScaleDrawable` (denoted by the `<scale>` root element) says that we should scale both height and width of the underlying drawable to 50% of the available space for the drawable, when the level is at its maximum (10000).

Note that you do not have to scale along both dimensions. If, for example, you kept `android:scaleWidth` but deleted `android:scaleHeight`, `setImageLevel()` would control the scaled width of the underlying image (provided via `android:drawable`) but not the height.

The `android:scaleGravity` attribute indicates where the scaled image should reside within the available space (the 10000 level, determined by the bounds of the `View` to which the drawable is applied). The value shown above, `center`, keeps the image centered within the available space, and shrinks or expands it around the center. A value of `left|top` would keep the image in the upper-left corner of the space, having the visual effect of moving the lower-right corner based upon the supplied level.

Clipping

Scaling proportionally reduces the height and/or width of an image. Clipping, on the other hand, chops off part of the height or width of the image.

```
<clip xmlns:android="http://schemas.android.com/apk/res/android"
  android:clipOrientation="horizontal"
  android:drawable="@drawable/btn_default_normal"
  android:gravity="left"/>
```

(from Drawable/ScaleClip/app/src/main/res/drawable/clip.xml)

In this sample `ClipDrawable` (indicated by the `<clip>` root element), we are going to allow the level to chop off part of the image indicated by the `android:drawable` attribute. Our `android:clipOrientation`, set to `horizontal`, means we are going to chop off part of the width (`vertical` would have us chop off part of the height). The amount that is going to be chopped off is the level you supply (e.g., `setImageLevel()`) divided by 10000. Hence, a level of 5000 will chop off 0.5 (a.k.a., 50%) of the image.

Where in the image the clipping occurs is determined by the `android:gravity` attribute. An `android:clipOrientation` of `horizontal` and an `android:gravity` of `left`, as in the sample drawable above, means that the left side of the image is retained, and the image will be clipped on the right. Specifying `right` instead of `left` would reverse that, clipping the image from the right, while `center` would clip equally from both sides. There are other gravity values as well, such as `top` and `bottom` values to be used with a `vertical` orientation.

Seeing It In Action

To see these effects, take a look at the `Drawable/ScaleClip` sample project. This is derived from an earlier example showing how to use `ViewPager` with `PagerTabStrip`. In that example, we had 10 tabs, each being a large `EditText` widget. In this example, we have 2 tabs, “Scale” and “Clip”, both using the same layout:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ImageView
    android:id="@+id/image"
    android:layout_width="150dp"
    android:layout_height="150dp"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="20dp"
    android:scaleType="fitXY"/>

  <SeekBar
    android:id="@+id/level"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_marginBottom="20dp"
    android:layout_marginLeft="20dp"
    android:layout_marginRight="20dp"
    android:max="10000"
    android:progress="10000"/>

</RelativeLayout>
```

(from Drawable/ScaleClip/app/src/main/res/layout/scaleclip.xml)

This is simply a `150dp` square `ImageView` towards the top of the screen and a `SeekBar` towards the bottom of the screen. The `SeekBar` will be used to control the level applied to a `ScaleDrawable` and `ClipDrawable`, which is why we have `android:max` set to `10000`. We also have our "progress" (original `SeekBar` value) set to `10000`, so the bar's thumb will be fully slid over to the right at the outset.

The fragments that we will use for the tabs both inherit from a common abstract `FragmentBase` class:

```
package com.commonsware.android.scaleclip;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.SeekBar;

abstract public class FragmentBase extends Fragment implements
```

```
    SeekBar.OnSeekBarChangeListener {
  abstract void setImageBackground(ImageView image);

  private ImageView image=null;

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    setRetainInstance(true);

    View result=inflater.inflate(R.layout.scaleclip, container, false);
    SeekBar bar=((SeekBar)result.findViewById(R.id.level));

    bar.setOnSeekBarChangeListener(this);
    image=(ImageView)result.findViewById(R.id.image);
    setImageBackground(image);
    image.setImageLevel(bar.getProgress());

    return(result);
  }

  @Override
  public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
    image.setImageLevel(progress);
  }

  @Override
  public void onStartTrackingTouch(SeekBar seekBar) {
    // no-op
  }

  @Override
  public void onStopTrackingTouch(SeekBar seekBar) {
    // no-op
  }
}
```

(from Drawable/ScaleClip/app/src/main/java/com/commonsware/android/scaleclip/FragmentBase.java)

In `onCreateView()`, we inflate the above layout file, hook up the fragment itself to be the listener for `SeekBar` change events, call the subclass' `setImageBackground()` method to populate the `ImageView` with an image, and set the `ImageView`'s level to be the initial value of the `SeekBar`. When the `SeekBar` value changes, our `onProgressChanged()` method will adjust the level.

The concrete subclasses — `ScaleFragment` and `ClipFragment` — simply populate the `ImageView` with the `ScaleDrawable` and `ClipDrawable` resources shown earlier in this section:

```
package com.commonsware.android.scaleclip;

import android.widget.ImageView;

public class ScaleFragment extends FragmentBase {
  @Override
  void setImageBackground(ImageView image) {
    image.setImageResource(R.drawable.scale);
  }
}
```

(from Drawable/ScaleClip/app/src/main/java/com/commonsware/android/scaleclip/ScaleFragment.java)

```
package com.commonsware.android.scaleclip;

import android.widget.ImageView;

public class ClipFragment extends FragmentBase {
  @Override
  void setImageBackground(ImageView image) {
    image.setImageResource(R.drawable.clip);
  }
}
```

(from Drawable/ScaleClip/app/src/main/java/com/commonsware/android/scaleclip/ClipFragment.java)

Those two drawables based their scaling and clipping on `res/drawable-xdpi/btn_default_normal.9.png`. This is a slightly-modified copy of the default button background, and is a nine-patch PNG file. We will discuss nine-patch PNG files later in this chapter — suffice it to say for now that it is a PNG file with rules about how it should be stretched.

Our scale tab starts off showing the full image:

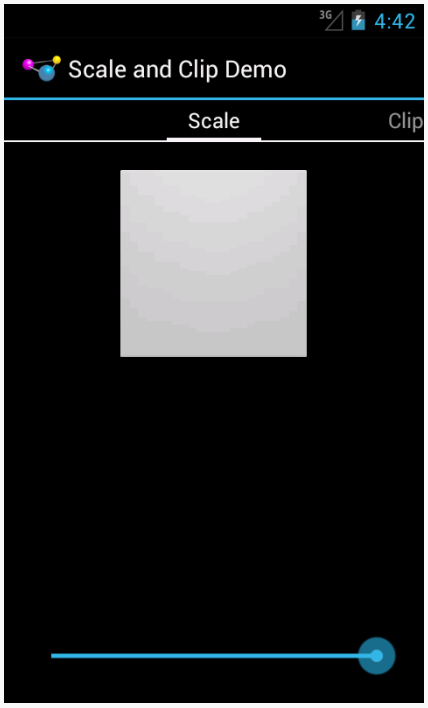


Figure 571: ScaleDrawable, Level of 10000

As we start sliding the `SeekBar` thumb to the left, the image shrinks progressively:

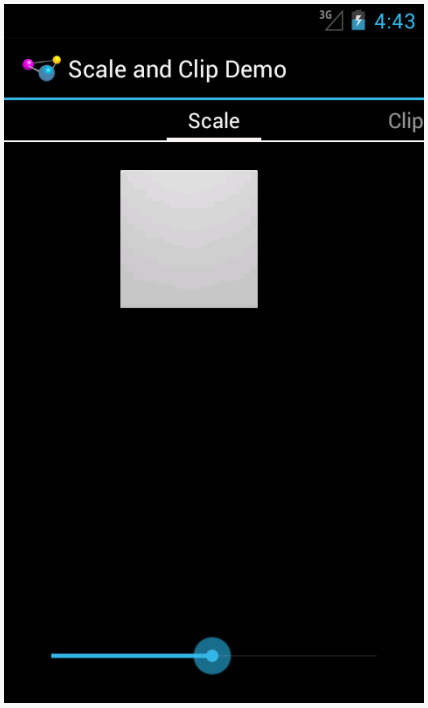


Figure 572: ScaleDrawable, Level of Approximately 5000

It eventually tends towards the 50% level specified in our `android:scaleHeight` and `android:scaleWidth` values:

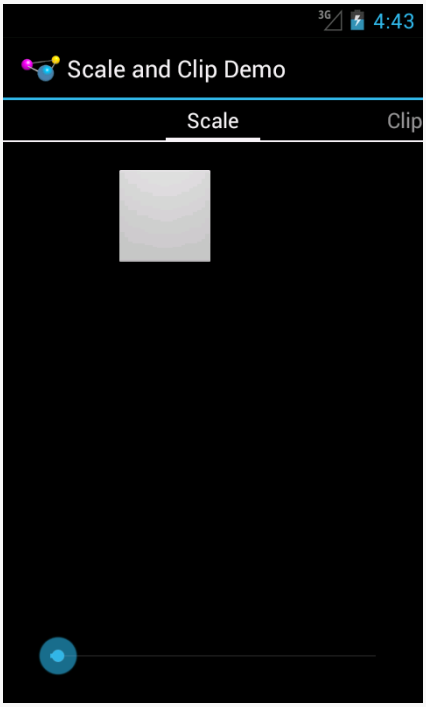


Figure 573: ScaleDrawable, Level of Approximately 100

Sliding it all the way to the left, though, causes the image to vanish.

The `ClipDrawable` starts off looking much like the `ScaleDrawable`:

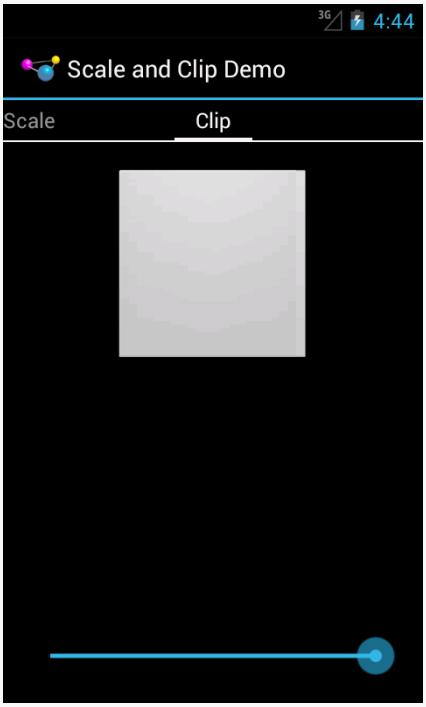


Figure 574: ClipDrawable, Level of 10000

As we slide the `SeekBar` to the left, the right side of the image gets clipped:

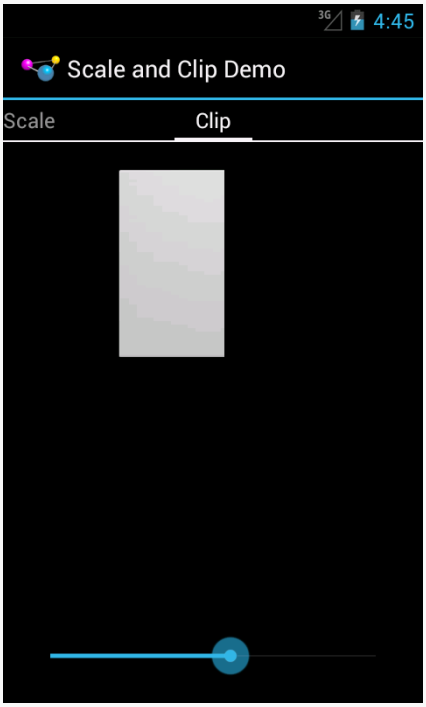


Figure 575: ClipDrawable, Level of Approximately 5000

InsetDrawable

An `InsetDrawable` allows you to apply insets on any side (or all sides) of some other drawable resource. The use case cited in the documentation is "This is used when a View needs a background that is smaller than the View's actual bounds". However, at the present time, nothing in the Android open source code uses this particular type of resource, or even the Java class.

In principle, though, you could have an XML drawable resource that looked like this:

```
<?xml version="1.0" encoding="utf-8"?>
<inset xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/something_or_another"
    android:insetLeft="20dp"
    android:insetTop="10dp" />
```

When used as the background for some `View`, for example, Android would pull in the `something_or_another` resource and effectively add `20dp` of left margin and `10dp` of top margin on the background when calculating its size and drawing it on the screen.

ShapeDrawable

`ShapeDrawable` is the original approach to implementing limited vector art on Android. It gives you what amounts to a very tiny subset of SVG, for creating simple vector art shapes.

The root element of a `ShapeDrawable` resource is `<shape>`, which may have child elements, along with attributes, to configure what gets rendered on the screen when the drawable is applied.

This section will review the elements and attributes available to you, with sample drawables (and screenshots) culled from the `Drawable/Shape` sample project.

This is a "sampler" project, designed to depict a number of `ShapeDrawables`. To accomplish this, we will use action bar tabs. Our activity (`MainActivity`) has a pair of static `int` arrays, one pointing at string resources to use for tab captions, the other pointing at corresponding drawable resources:

```
package com.commonsware.android.shape;

import android.app.ActionBar;
import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.widget.ImageView;

public class MainActivity extends Activity implements TabListener {
  private static final int TABS[]= { R.string.solid, R.string.gradient,
      R.string.border, R.string.rounded, R.string.ring,
      R.string.layered };
  private static final int DRAWABLES[]= { R.drawable.rectangle,
      R.drawable.gradient, R.drawable.border, R.drawable.rounded,
      R.drawable.ring, R.drawable.layered };
  private ImageView image=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    image=(ImageView)findViewById(R.id.image);
```

```
    ActionBar bar=getActionBar();
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    for (int i=0; i < TABS.length; i++) {
      bar.addTab(bar.newTab().setText(getString(TABS[i]))
                 .setTabListener(this));
    }
  }

  @Override
  public void onTabSelected(Tab tab, FragmentTransaction ft) {
    image.setImageResource(DRAWABLES[tab.getPosition()]);
  }

  @Override
  public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }

  @Override
  public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }
}
```

(from Drawable/Shape/app/src/main/java/com/commonsware/android/shape/MainActivity.java)

In `onCreate()`, we toggle the `ActionBar` into tab-navigation mode, then iterate over the arrays and add one tab per element.

Our layout is an `ImageView`, named `image`, centered on the screen, taking up 80% of the horizontal space, plus has `20dp` of top and bottom margin:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/LinearLayout1"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal"
  android:gravity="center"
  android:weightSum="10">

    <ImageView
      android:id="@+id/image"
      android:src="@drawable/rectangle"
      android:layout_width="0dp"
      android:layout_height="match_parent"
```

```
      android:layout_marginTop="20dp"
      android:layout_marginBottom="20dp"
      android:layout_gravity="center"
      android:layout_weight="8"/>

</LinearLayout>
```

(from Drawable/Shape/app/src/main/res/layout/activity_main.xml)

In our activity's `onTabSelected()` — implemented because the activity is the `TabListener` for our tabs — we get the position of our tab and fill in the appropriate drawable into the `ImageView`.

Given that, let's take a look at how to construct a `ShapeDrawable`, along with some sample drawables.

<shape>

Your root element, not surprisingly, is `<shape>`.

The primary thing that you will define on the `<shape>` element is the redundantly-named `android:shape` attribute, to define what sort of shape you want:

- `line` (a shape with no interior)
- `oval` (also for ellipses)
- `rectangle` (including rounded rectangles)
- `ring` (for partially-filled circles)

There are some other attributes available on `<shape>` for a `ring`, which we will examine later in this chapter.

<solid>

Your shape will usually require some sort of fill, to say what color goes in the shape. There are two types of fills: solid and gradient.

For a solid fill, add a `<solid>` child element to the `<shape>`, with an `android:color` attribute indicating what color to use. As with most places in Android, this can either be a literal color or a reference to a color resource.

So, for example, we can specify a solid red rectangle as:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">
  <solid android:color="#FFAA0000"/>
</shape>
```

(from Drawable/Shape/app/src/main/res/drawable/rectangle.xml)

This gives us the following visual result:

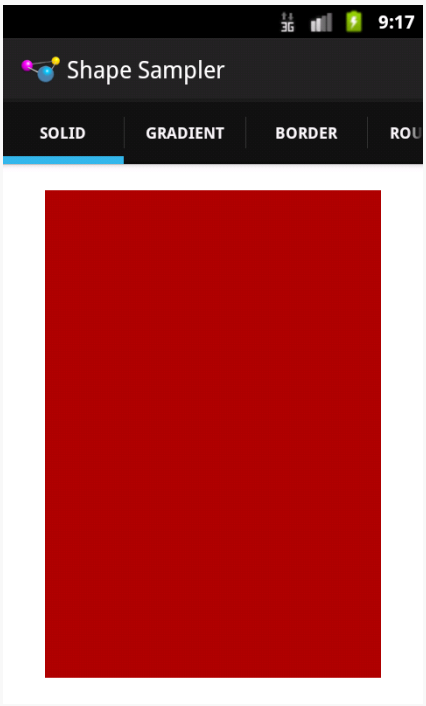


Figure 576: ShapeDrawable, Solid Red Rectangle

<gradient>

Your alternative fill is a gradient. The nice thing about gradients with `ShapeDrawable` is that they are generated at runtime from the specifications in the `ShapeDrawable`, and therefore will be smooth. Gradients that appear in PNG files and the like, if stretched, will tend to have a banding effect.

Gradient fills are defined via a `<gradient>` child element of the `<shape>` element.

The simplest way to set up a gradient is to use three attributes:

- `android:startColor` and `android:endColor`, to specify the starting and ending colors of the gradient, respectively, and
- `android:angle`, to specify what direction the gradient "flows" in

The angle must be a multiple of 45 degrees. 0 degrees is left-to-right, 90 degrees is bottom-to-top, 180 degrees is right-to-left, and 270 degrees is top-to-bottom.

So, for example, we could change our rectangle to have a gradient fill, from red to blue, with red at the top, via:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <gradient
    android:angle="270"
    android:endColor="#FF0000FF"
    android:startColor="#FFFF0000"/>

</shape>
```

(from Drawable/Shape/app/src/main/res/drawable/gradient.xml)

That gives us:

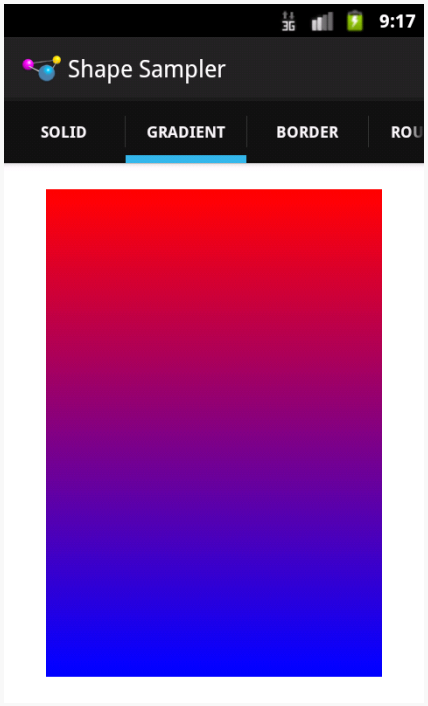


Figure 577: ShapeDrawable, Gradient Fill Rectangle

We will examine some other gradient options in the section on rings, later in this chapter.

<stroke>

If you want a separate color for a border around your shape, you can use the `<stroke>` element, as a child of the `<shape>` element, to configure one.

There are four attributes that you can declare. The two that you will probably always use are `android:color` (to indicate the color of the border) and `android:width` (to indicate the thickness of the border). By default, using just those two will give you a solid line around the edge of your shape.

If you would prefer a dashed border, you can add in `android:dashWidth` (to indicate how long each dash segment should be) and `android:dashGap` (to indicate how long the gaps between dash segments should be).

So, for example, we can add a dashed border to our gradient rectangle via a suitable `<stroke>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <gradient
    android:angle="270"
    android:endColor="#FF0000FF"
    android:startColor="#FFFF0000"/>

  <stroke
    android:width="2dp"
    android:dashGap="4dp"
    android:dashWidth="20dp"
    android:color="#FF000000"/>

</shape>
```

(from Drawable/Shape/app/src/main/res/drawable/border.xml)

This gives us:

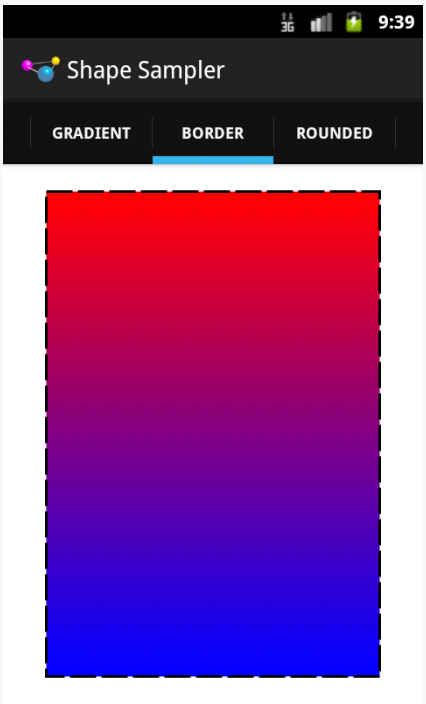


Figure 578: ShapeDrawable, Gradient Fill Rectangle with Dashed Border

<corners>

If we are implementing a `rectangle` shape, but we really want it to be a rounded rectangle, we can add a `<corners>` element as a child of the `<shape>` element. You can specify the radius to apply to the corners, either for all corners (e.g., `android:radius`), or for individual corners (e.g., `android:topLeftRadius`). Here, "radius" basically means the size of the circle that should implement the corner, where a radius of `0dp` would indicate the default square corner.

So, if we wanted to add rounded corners to our gradient-filled, dash-outlined rectangle, we could use this:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <gradient
    android:angle="270"
    android:endColor="#FF0000FF"
    android:startColor="#FFFF0000"/>

  <stroke
    android:dashGap="4dp"
    android:dashWidth="20dp"
    android:width="2dp"
    android:color="#FF000000"/>

  <corners android:radius="8dp"/>

</shape>
```

(from Drawable/Shape/app/src/main/res/drawable/rounded.xml)

This gives us the following:

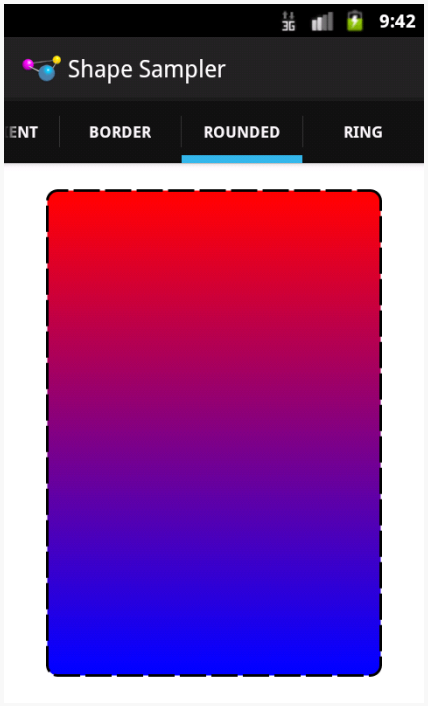


Figure 579: ShapeDrawable, Gradient Fill Rounded Rectangle with Dashed Border

<padding> and <size>

There also `<padding>` and `<size>` elements that you can add, that specify padding to put on the various sizes and the overall size of the drawable. More often than not, you would actually handle this on the `ImageView` or other widget that is using your drawable, but if you would prefer to define those things in the drawable itself, you are welcome to do so.

Put a Ring On It

Rings are a bit more complicated, in large part because they are not completely filled. With a ring, the "fill" is filling what goes in the ring itself, not the "hole" in the center of the ring. This means that we need to teach Android more about how that "hole" is supposed to be set up.

To do that, we need to provide two pieces of information:

1. How big the inner radius should be, where by "inner radius" Android means "the radius of the hole"
2. How thick the ring should be

The ring will then be drawn based upon that inner radius and thickness.

You might wonder, "well, where does the size of the actual drawable come into play?" After all, if we specify an inner radius of `20dp` and a thickness of `10dp`, that would give us an outer radius of `30dp`, for a total width of `60dp`... regardless of how big the actual drawable is.

And that is completely correct.

However, for both the inner radius and the thickness, you have two choices of how to specify their values:

1. As actual sizes (dimensions or references to dimension resources)
2. As ratios to the overall drawable width (defined by `<size>` or the widget that is using the drawable)

This gives us four total attributes to choose from, to be placed on the `<shape>` element for `ring` drawables:

1. `android:innerRadius`
2. `android:innerRadiusRatio`
3. `android:thickness`
4. `android:thicknessRatio`

Therefore, if you want the ring's size to be based on the size of the drawable, you would use `innerRadiusRatio`, `thicknessRatio`, or both.

The other thing about rings is that they are round. Hence, a default linear gradient fill — going from one side of the drawable to another – may not be what you really want. You can control the type of gradient fill to use via the `android:type` attribute on the `<gradient>` element. There are three possible values:

1. `linear` (the default behavior)
2. `radial`, where the gradient starts from the center (or another point that you define) and changes color from that center to the edges
3. `sweep`, where the gradient revolves clockwise in a circle, starting from whatever `android:angle` you specify (or `0`, meaning "east", as the default)

So, for example, take a look at the following `ShapeDrawable`:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  android:innerRadiusRatio="3"
  android:shape="ring"
  android:thickness="15dp"
  android:useLevel="false">

  <gradient
    android:centerColor="#4c737373"
    android:endColor="#ff9933CC"
    android:startColor="#4c737373"
    android:type="sweep"/>

</shape>
```

(from Drawable/Shape/app/src/main/res/drawable/ring.xml)

Here, we:

- Declare that our shape is a `ring`
- Indicate that the distance between the inner radius and the outer radius of the ring should be `15dp`
- Indicate that there is a 3:1 ratio between the width of the image and the radius of the "hole" in the ring
- Indicate that the fill should be a gradient that sweeps clockwise from the default angle of 0
- Indicate that the first half of the gradient (start to center) should remain a constant color
- Indicate that the second half of the gradient (center to end) should change color from gray to purple

We also have `android:useLevel="false"` in the `<shape>` element. For unknown reasons, this is required for rings but not for other types of shapes.

This gives us:

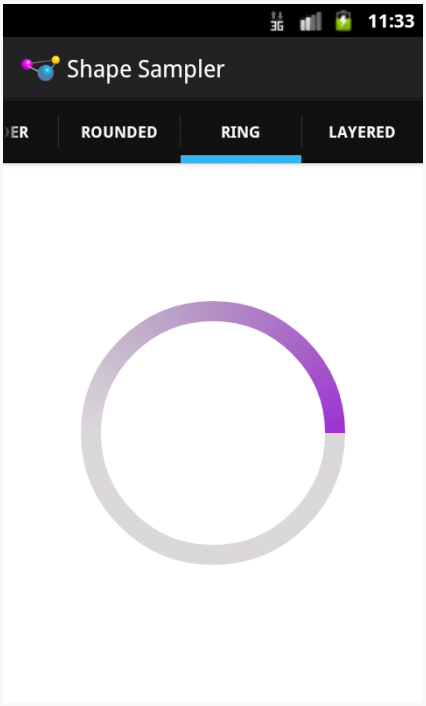


Figure 580: ShapeDrawable, Ring with Gradient Fill

BitmapDrawable

Having an XML drawable format named `BitmapDrawable` may seem like a contradiction in terms. However, `BitmapDrawable` is not an XML representation of a bitmap, but rather an XML representation of operations to perform on an actual bitmap.

The big thing that `BitmapDrawable` offers is `android:tileMode`, which turns a single bitmap into a repeating bitmap. The bitmap is tiled, horizontally and vertically, using a tiling mode that you specify.

This is demonstrated in the [Drawable/TileMode] sample project.

Our activity's layout is just a `LinearLayout`, set to fill the screen:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/widget"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
```

```
  android:orientation="horizontal">

</LinearLayout>
```

(from Drawable/TileMode/app/src/main/res/layout/activity_main.xml)

Our activity populates action bar tabs, where it applies a particular background image to the `LinearLayout` (known as `R.id.widget`) based on the selected tab:

```
package com.commonsware.android.tilemode;

import android.app.ActionBar;
import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends Activity implements TabListener {
  private static final int TABS[]= { R.string._default, R.string.clamp,
      R.string.repeat, R.string.mirror };
  private static final int DRAWABLES[]= { R.drawable._default,
      R.drawable.clamp, R.drawable.repeat, R.drawable.mirror };
  private View widget=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    widget=findViewById(R.id.widget);

    ActionBar bar=getActionBar();
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    for (int i=0; i < TABS.length; i++) {
      bar.addTab(bar.newTab().setText(getString(TABS[i]))
                    .setTabListener(this));
    }
  }

  @Override
  public void onTabSelected(Tab tab, FragmentTransaction ft) {
    widget.setBackgroundResource(DRAWABLES[tab.getPosition()]);
  }

  @Override
```

```
  public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }

  @Override
  public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }
}
```

(from Drawable/TileMode/app/src/main/java/com/commonsware/android/tilemode/MainActivity.java)

The `res/drawable/_default.xml` resource, used on the first tab, is an unadorned `BitmapDrawable` resource, where our `<bitmap>` element simply has an `android:src` attribute pointing to a bitmap to be used for this `BitmapDrawable`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"/>
```

(from Drawable/TileMode/app/src/main/res/drawable/_default.xml)

Since we have not specified a tile mode, the image is stretched to fill the size of our `LinearLayout` when serving as its background:

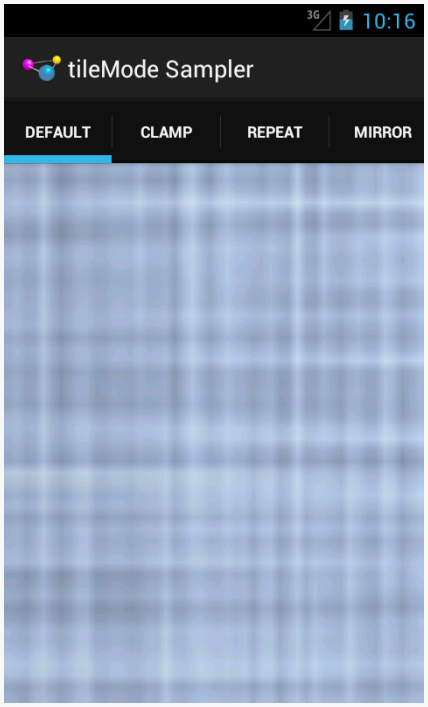


Figure 581: BitmapDrawable, Without android:tileMode

The `res/drawable/clamp.xml` resource, used on the second tab, adds `android:tileMode="clamp"`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"
  android:tileMode="clamp"/>
```

(from Drawable/TileMode/app/src/main/res/drawable/clamp.xml)

This causes the right-most column of pixels and the bottom-most row of pixels to be repeated to fill the available space:

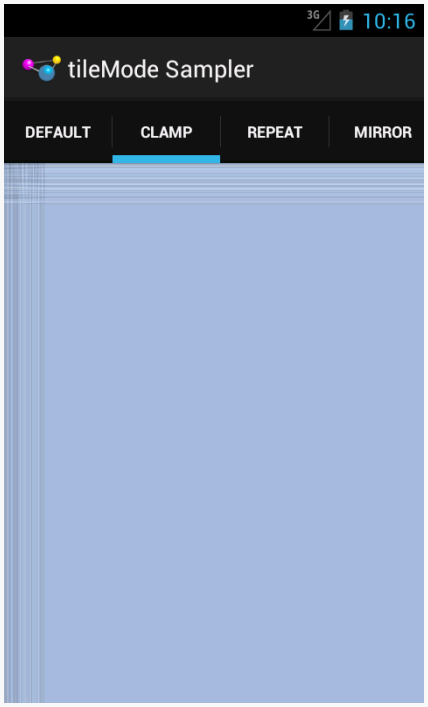


Figure 582: BitmapDrawable, Clamped

Zooming in on the upper-left portion of our `LinearLayout` demonstrates this:

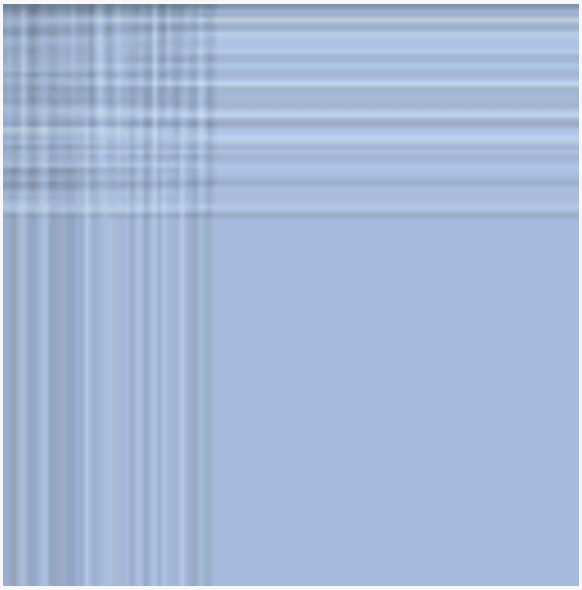


Figure 583: Portion of BitmapDrawable, Clamped

The `res/drawable/repeat.xml` resource, used on the third tab, employs `android:tileMode="repeat"`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"
  android:tileMode="repeat"/>
```

(from Drawable/TileMode/app/src/main/res/drawable/repeat.xml)

Here, the image is simply repeated *in toto* to fill the available space, rather than only its lower-right edges:

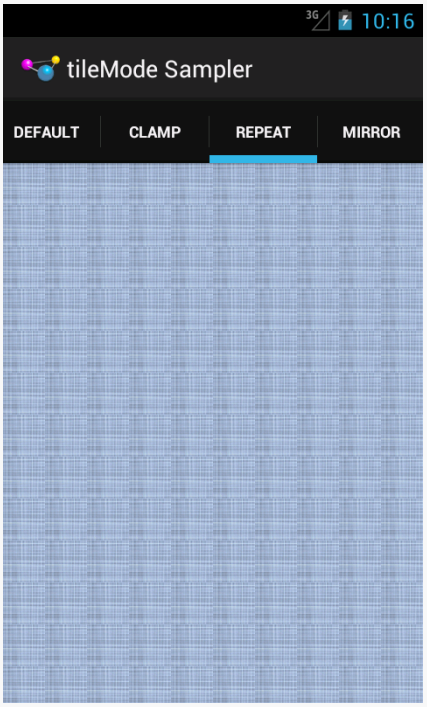


Figure 584: BitmapDrawable, Repeated

Zooming in on an arbitrary chunk of the `LinearLayout` shows this effect:

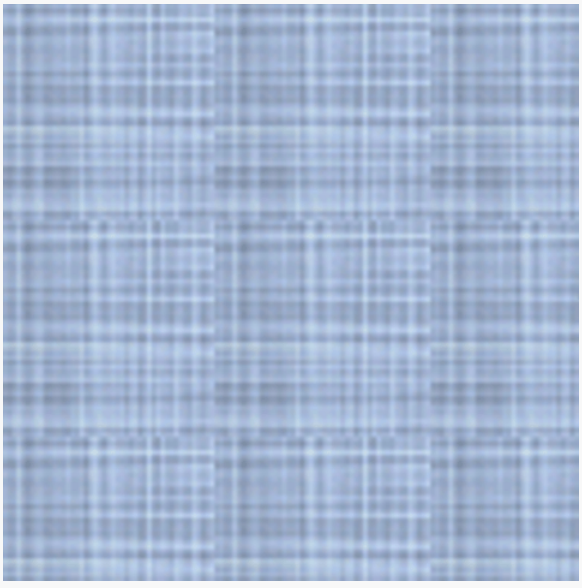


Figure 585: Portion of BitmapDrawable, Repeated

The `res/drawable/mirror.xml` resource, used on the fourth tab, uses `android:tileMode="mirror"`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"
  android:tileMode="mirror"/>
```

(from Drawable/TileMode/app/src/main/res/drawable/mirror.xml)

Here, the image is repeated, but alternately mirrored along the repeating axis. So, it is flipped horizontally for each repeat along the horizontal axis, and it is flipped vertically for each repeat along the vertical axis:

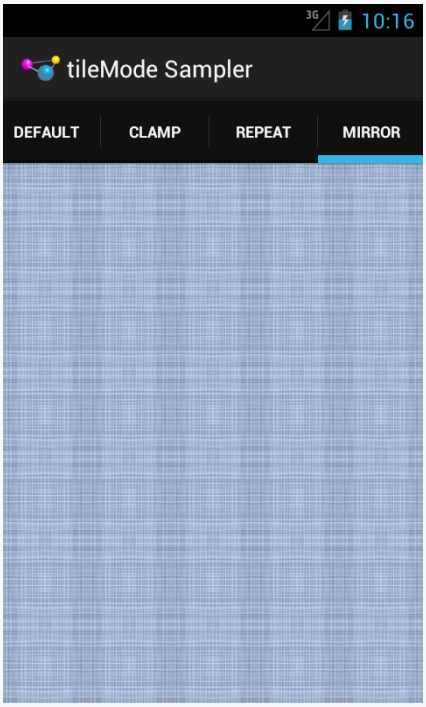


Figure 586: BitmapDrawable, Mirrored

Zooming in on an arbitrary chunk of the `LinearLayout` shows this effect:

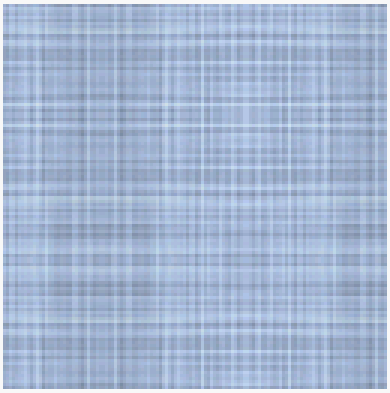


Figure 587: Portion of BitmapDrawable, Mirrored

Composite Drawables

Let's say that we wanted to have a pair of `ShapeDrawable` images, one superimposed on another. Since a single `ShapeDrawable` defines only one shape, we would need something else to assist with stacking the images.

One possibility would be to use a `LayerDrawable`, creating three total resources:

1. The first `ShapeDrawable`, in its own resource file
2. The second `ShapeDrawable`, in its own resource file
3. The `LayerDrawable`, holding references to the two `ShapeDrawable` resources

And this will certainly work. But you have an alternative: put all of it into a single drawable resource.

An `android:drawable` attribute in an `<item>` element can be replaced by child elements representing another drawable structure. Hence, rather than having a `LayerDrawable` with two `<item>` elements pointing to other drawable resources, we could have those same `<item>` elements *contain* the other drawable XML structures, and thereby cut our number of files from 3 to 1.

For example, we could have something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

  <item>
    <shape android:shape="rectangle">
      <gradient
```

```
        android:angle="270"
        android:endColor="#FF0000FF"
        android:startColor="#FFFF0000"/>

      <stroke
        android:dashGap="4dp"
        android:dashWidth="20dp"
        android:width="2dp"
        android:color="#FF000000"/>

      <corners android:radius="8dp"/>
    </shape>
  </item>
  <item>
    <shape
      android:innerRadiusRatio="3"
      android:shape="ring"
      android:thickness="15dp"
      android:useLevel="false">
      <gradient
        android:endColor="#FFFFFFFF"
        android:startColor="#ff000000"
        android:type="sweep"/>
    </shape>
  </item>

</layer-list>
```

(from Drawable/Shape/app/src/main/res/drawable/layered.xml)

This is a `LayerDrawable`, layering two `ShapeDrawable` structures. The first `ShapeDrawable` is our dash-bordered, gradient-filled, rounded rectangle from before. The second `ShapeDrawable` is a `ring` with a simple gradient sweep fill, from black to white.

This gives us:

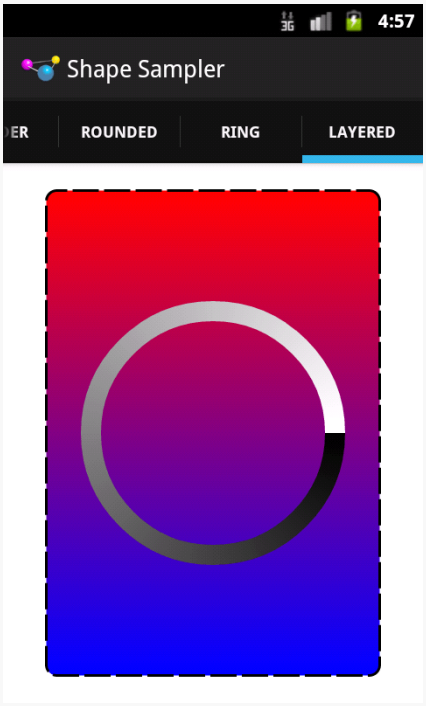


Figure 588: Composite Drawable

Hence, any of the drawable XML structures other than `ShapeDrawable` can, in their `<item>` elements, hold any drawable XML structure, instead of pointing to another separate resource.

Android uses this trick as well. For example, the stock `ProgressBar` image is based off of a `LayerDrawable` wrapped around three `ShapeDrawable` structures:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->
```

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@android:id/background">
        <shape>
            <corners android:radius="5dip" />
            <gradient
                    android:startColor="#ff9d9e9d"
                    android:centerColor="#ff5a5d5a"
                    android:centerY="0.75"
                    android:endColor="#ff747674"
                    android:angle="270"
            />
        </shape>
    </item>

    <item android:id="@android:id/secondaryProgress">
        <clip>
            <shape>
                <corners android:radius="5dip" />
                <gradient
                        android:startColor="#80ffd300"
                        android:centerColor="#80ffb600"
                        android:centerY="0.75"
                        android:endColor="#a0ffcb00"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>

    <item android:id="@android:id/progress">
        <clip>
            <shape>
                <corners android:radius="5dip" />
                <gradient
                        android:startColor="#ffffd300"
                        android:centerColor="#ffffb600"
                        android:centerY="0.75"
                        android:endColor="#ffffcb00"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>

</layer-list>
```

We will get into how this works with a `ProgressBar` in a separate chapter.

A Stitch In Time Saves Nine

Most of the types of non-traditional drawable resources you can create in Android are described in XML... but not all.

As you read through the Android documentation, you no doubt ran into references to "nine-patch" or "9-patch" and wondered what Android had to do with quilting. Rest assured, you will not need to take up needlework to be an effective Android developer.

If, however, you are looking to create backgrounds for resizable widgets, like a `Button`, you may wish to work with nine-patch images.

As the Android documentation states, a nine-patch is "a PNG image in which you define stretchable sections that Android will resize to fit the object at display time to accommodate variable sized sections, such as text strings". By using a specially-created PNG file, Android can avoid trying to use vector-based formats (e.g., `ShapeDrawable`) and their associated overhead when trying to create a background at runtime. Yet, at the same time, Android can still resize the background to handle whatever you want to put inside of it, such as the text of a `Button`.

In this section, we will cover some of the basics of nine-patch graphics, including how to customize and apply them to your own Android layouts.

Note that nine-patch PNG files, while they provide stretching rules, are still somewhat dependent upon density. You may wish to have different versions of your nine-patch PNG files for different densities, and therefore these images should be put in density-specific resource directories (e.g., `res/drawable-hdpi/`).

The Name and the Border

Nine-patch graphics are PNG files whose names end in `.9.png`. This means they can be edited using normal graphics tools, but Android knows to apply nine-patch rules to their use.

What makes a nine-patch graphic different than an ordinary PNG is a one-pixel-wide border surrounding the image. When drawn, Android will remove that border, showing only the stretched rendition of what lies inside the border. The border is

used as a control channel, providing instructions to Android for how to deal with stretching the image to fit its contents.

Padding and the Box

Along the right and bottom sides, you can draw one-pixel-wide black lines to indicate the "padding box". Android will stretch the image such that the contents of the widget will fit inside that padding box.

For example, suppose we are using a nine-patch as the background of a `Button`. When you set the text to appear in the button (e.g., "Hello, world!"), Android will compute the size of that text, in terms of width and height in pixels. Then, it will stretch the nine-patch image such that the text will reside inside the padding box. What lies outside the padding box forms the border of the button, typically a rounded rectangle of some form.

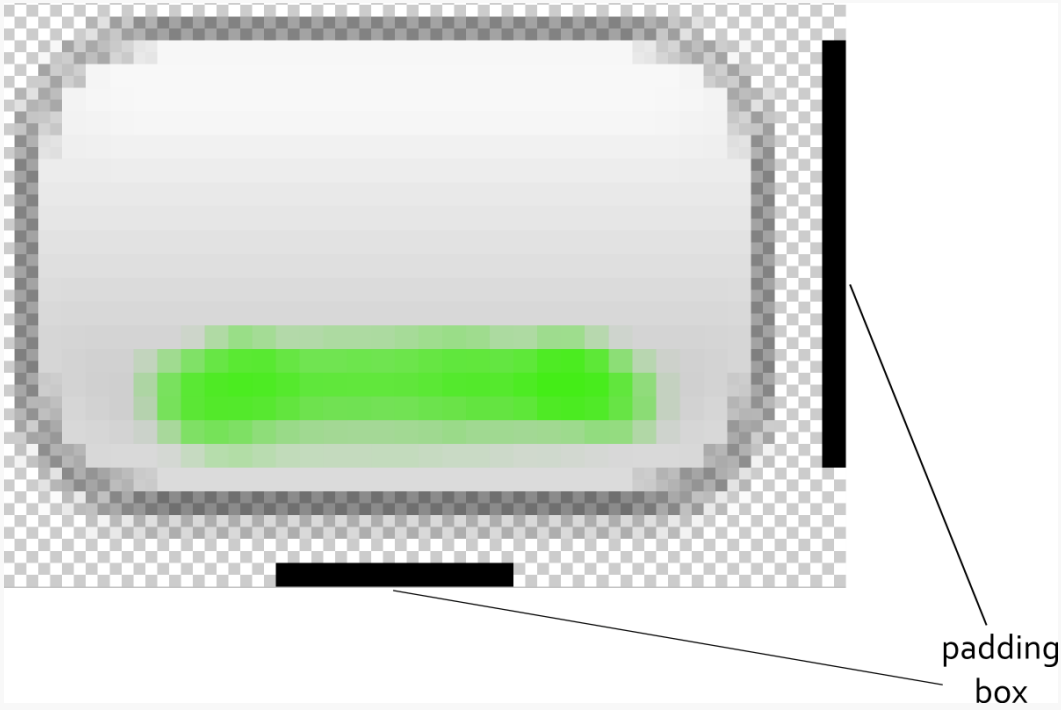


Figure 589: The padding box, as shown by a set of control lines to the right and bottom of the stretchable image

Stretch Zones

To tell Android where on the image to actually do the stretching, draw one-pixel-wide black lines on the top and left sides of the image. Android will scale the graphic only in those areas — areas outside the stretch zones are not stretched.

Perhaps the most common pattern is the center-stretch, where the middle portions of the image on both axes are considered stretchable, but the edges are not:

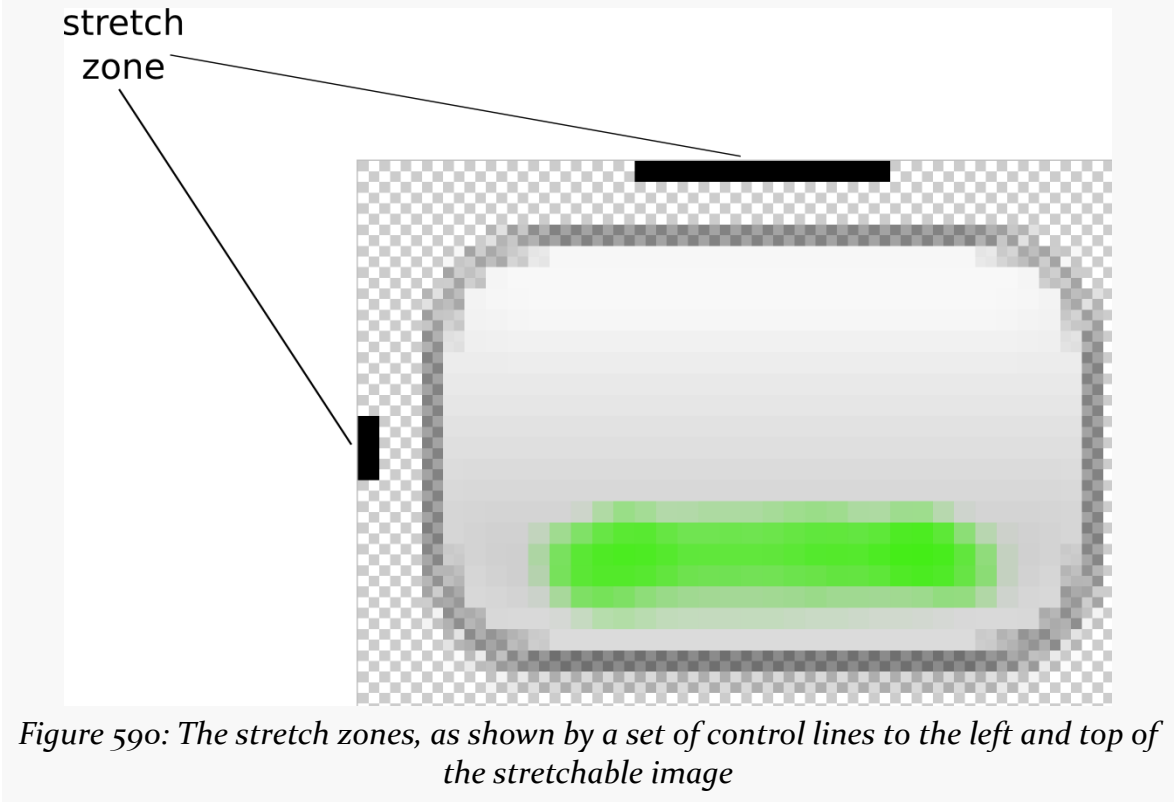


Figure 590: The stretch zones, as shown by a set of control lines to the left and top of the stretchable image

Here, the stretch zones will be stretched just enough for the contents to fit in the padding box. The edges of the graphic are left unstretched.

Some additional rules to bear in mind:

1. If you have multiple discrete stretch zones along an axis (e.g., two zones separated by whitespace), Android will stretch both of them but keep them in their current proportions. So, if the first zone is twice as wide as the second zone in the original graphic, the first zone will be twice as wide as the second zone in the stretched graphic.

2. If you leave out the control lines for the padding box, it is assumed that the padding box and the stretch zones are one and the same.

Tooling

Android Studio has a nine-patch PNG editor, based on the now-discontinued `draw9patch` utility used previously by Android app developers. To work with this tool:

- If you have a regular PNG that you want to convert into a nine-patch PNG, right-click over it and choose "Create 9-patch file" from the context menu
- If you already have a nine-patch PNG (with the `.9.png` extension), or you created one using "Create 9-patch file", just double-click on the image to bring it up in a nine-patch editor

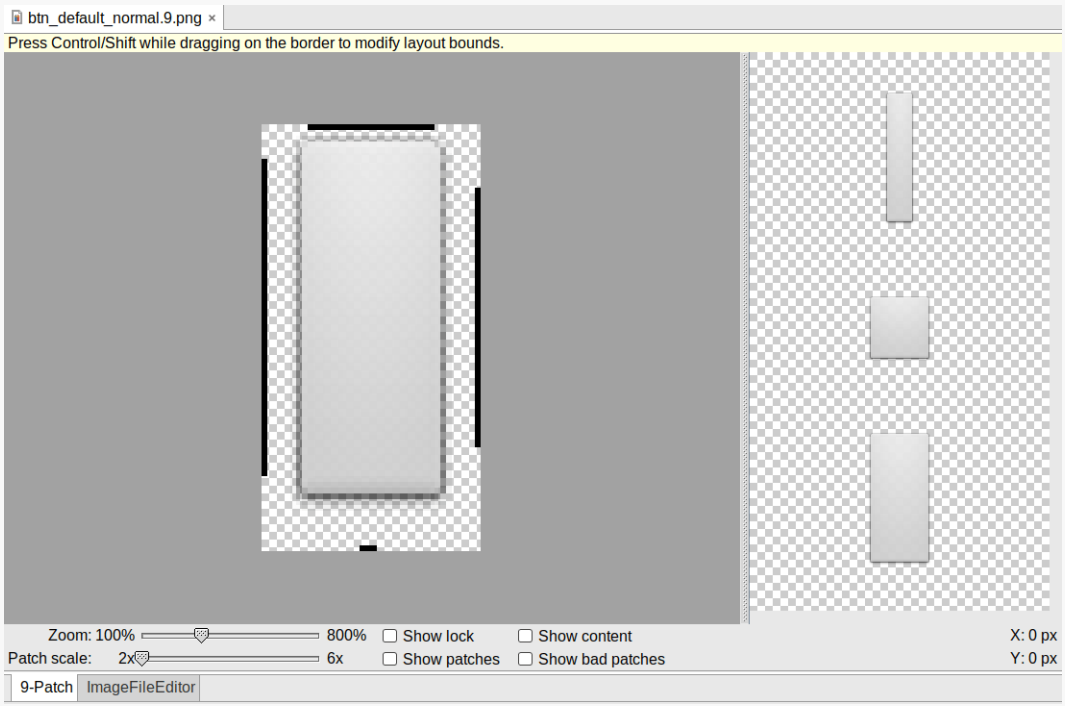


Figure 591: Android Studio, Editing a Nine-Patch PNG

While a regular graphics editor would allow you to draw any color on any pixel, the nine-patch editor limits you to drawing or erasing pixels in the control area:

- Set a pixel (making it black) by clicking on it
- Clear a pixel by shift-clicking on it

If you attempt to draw inside the main image area itself, you will be blocked.

On the right, you will see samples of the image in various stretched sizes, so you can see the impact as you change the stretchable zones and padding box.

While this is convenient for working with the nine-patch nature of the image, you will still need some other graphics editor to create or modify the body of the image itself. For example, the image shown above, from the `Drawable/NinePatch` project, is a modified version of a nine-patch graphic from the SDK's `ApiDemos`, where the GIMP was used to add the neon green stripe across the bottom portion of the image.

Using Nine-Patch Images

Nine-patch images are most commonly used as backgrounds, as illustrated by the following layout from the `Drawable/NinePatch` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
  >
    <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Horizontal:"
      />
      <SeekBar android:id="@+id/horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
```

```
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Vertical:"
      />
      <SeekBar android:id="@+id/vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
  </TableLayout>
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <Button android:id="@+id/resize"
      android:layout_width="96px"
      android:layout_height="96px"
      android:text="Hi!"
      android:textSize="10sp"
      android:background="@drawable/button"
    />
  </LinearLayout>
</LinearLayout>
```

(from Drawable/NinePatch/app/src/main/res/layout/main.xml)

Here, we have two `SeekBar` widgets, labeled for the horizontal and vertical axes, plus a `Button` set up with our nine-patch graphic as its background (`android:background = "@drawable/button"`).

The `NinePatchDemo` activity then uses the two `SeekBar` widgets to let the user control how large the button should be drawn on-screen, starting from an initial size of `64px` square:

```
package com.commonsware.android.ninepatch;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.widget.SeekBar;

public class NinePatchDemo extends Activity {
  SeekBar horizontal=null;
```

```
  SeekBar vertical=null;
  View thingToResize=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    thingToResize=findViewById(R.id.resize);

    horizontal=(SeekBar)findViewById(R.id.horizontal);
    vertical=(SeekBar)findViewById(R.id.vertical);

    horizontal.setMax(144); // 240 less 96 starting size
    vertical.setMax(144);  // keep it square @ max

    horizontal.setOnSeekBarChangeListener(h);
    vertical.setOnSeekBarChangeListener(v);
  }

  SeekBar.OnSeekBarChangeListener h=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                  int progress,
                                  boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(64+progress,
                                                                   old.height);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };

  SeekBar.OnSeekBarChangeListener v=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                  int progress,
                                  boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(old.width,
                                                                   64+progress);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };
}
```

(from Drawable/NinePatch/app/src/main/java/com/commonsware/android/ninepatch/NinePatchDemo.java)

The result is an application that can be used much like the right pane of `draw9patch`, to see how the nine-patch graphic looks on an actual device or emulator in various sizes:

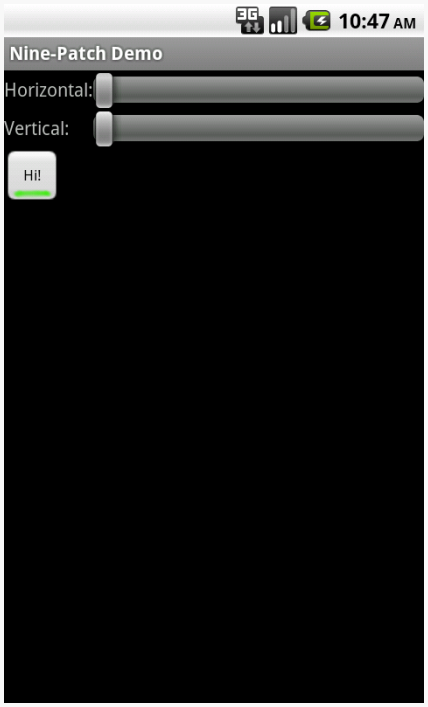


Figure 592: The NinePatch sample project, in its initial state

Figure 593: The NinePatch sample project, after making it bigger horizontally

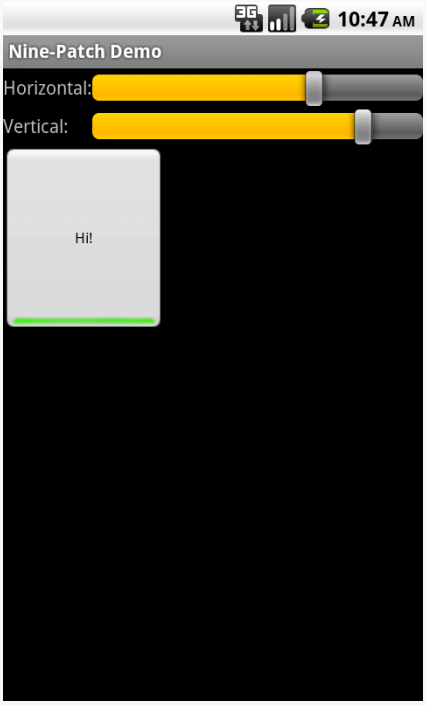


Figure 594: The NinePatch sample application, after making it bigger in both dimensions

Mapping with Maps V2

One of Google's most popular services — after search, of course – is Google Maps, where you can find everything from the nearest pizza parlor to directions from New York City to San Francisco (only 2,905 miles!) to street views and satellite imagery.

Android has had mapping capability from the beginning, with an API available to us as developers to bake maps into our apps. However, as we will see shortly, that original API was getting a bit stale.

In December 2012, Google released a long-awaited update to the mapping capabilities available to Android app developers. The original mapping solution, now known as the Maps V1, worked but had serious limitations. The new mapping solution, known as Maps V2, offers greater power and greater ease of handling common situations, though it too has its rough edges.

Prerequisites

Understanding this chapter requires that you have read the core chapters, along with the chapter on drawables. Also, one of the samples involves location tracking, and another of the samples involves the use of the animator framework.

One section involves the use of Picasso, covered in the chapter on Internet access.

This chapter also makes the occasional reference back to Maps V1 for comparisons, mostly for the benefit of developers already familiar with Maps V1 and looking to migrate to Maps V2. However, prior experience with Maps V1 is not necessary to understand this chapter.

A Brief History of Mapping on Android

Back in the dawn of Android, we were given the Maps SDK add-on. This would allow us to load a firmware-hosted mapping library into our applications, then embed maps into our activities, by means of a `MapView` widget.

And it worked.

More importantly, from the standpoint of users, the results from our apps were visually indistinguishable from the built-in Maps application available on devices that had the Maps SDK add-on.

This was the case through most of 2009. Eventually, though, the Google Maps team wanted to update the Maps application... but, for whatever reason, the decision was made to *not* update the Maps SDK add-on as well. At this point, the Google Maps team effectively forked the Maps SDK add-on, causing the Maps application to diverge from what other Android app developers could deliver. Over time, this feature gap became quite pronounced.

The release of Android 3.0 in early 2011 compounded the problems. Now, we needed to consider using fragments to help manage our code and deliver solutions to all screen sizes. Alas, while we could add maps to our fragments, we could only do so on API Level 11 or higher — the fragments backport from the Android Support package did not work with the Maps SDK add-on.

The release of Maps V2 helped all of this significantly. Now we have proper map support for native and backported versions of the fragment framework. We also have a look and feel that is closer to what the Maps application itself supports. While we still cannot reach feature parity with the Maps application, our SDK apps can at least look like they belong on the same device as the Maps application.

More importantly, as of the time of this writing, Maps V1 is no longer an option for new developers. Those who already have Maps V1 API keys can use Maps V1, but no new Maps V1 API keys are being offered. That leaves you with either using Maps V2 or some alternative mapping solution.

Where You Can Use Maps V2

Many devices will be able to use Maps V2... but not all. Notably:

- Devices need to support OpenGL ES 2.0+, to handle the new vector-based tiles that Maps V2 uses. Over 99% of Android devices in use today that support the Play Store (or its "Android Market" predecessor) also support OpenGL ES 2.0+.
- Devices will need an update to the Play Services Framework that accompanies the Play Store. Devices that do not have the Play Store — either because they are forever stuck on the old Android Market or, like the Kindle Fire, never had Play Store support in the first place — will be unable to use Maps V2.

Later in this chapter, we will look at other mapping libraries that you could use instead of either of Google's mapping solutions.

Licensing Terms for Maps V2

As with the original Maps SDK add-on, to use Maps V2, you must agree to a terms of service agreement to be authorized to embed Google Maps within your application. If you intend to use Maps V2, you should review these terms closely, as they place many restrictions on developers. The most notorious of these is that you cannot use Maps V2 to create an application that offers "real time navigation or route guidance, including but not limited to turn-by-turn route guidance that is synchronized to the position of a user's sensor-enabled device."

If you find these terms to be an issue for your application, you may need to consider alternative mapping solutions.

What You Need to Start

If you wish to use Maps V2 in one or more of your Android applications, this section will outline what you need to get started.

Your Signing Key Fingerprint(s)

As with the legacy Maps SDK add-on, you will need fingerprints of your app signing keys, to tie your apps to your Google account and the API keys you will be generating. However, unlike the legacy Maps SDK add-on, the fingerprints you will be using will be created using the SHA-1 hash algorithm, rather than MD5.

First, you will need to know where the keystore is for your signing key. For a production keystore that you created yourself for your production apps, you should know where it is located already. For the debug keystore, used by default during development, the location is dependent upon operating system:

- macOS and Linux: `~/.android/debug.keystore`
- Windows XP: `C:\Documents and Settings\$USER\.android\debug.keystore`
- Windows Vista and Windows 7: `C:\Users\$USER\.android\debug.keystore`

(where `$USER` is your Windows user name)

You will then need to run the `keytool` command, to dump information related to this keystore. The `keytool` command is in your Java SDK, not the Android SDK. You will need to run this from a command line (e.g., Command Prompt in Windows). The specific command to run is:

```
keytool -list -v -keystore ... -alias androiddebugkey -storepass android -keypass
android
```

where the `...` is replaced by the path to your debug keystore, enclosed in quotation marks if the path contains spaces. For your production keystore, you would supply your own alias and passwords.

This should emit output akin to:

```
Alias name: androiddebugkey
Creation date: Aug 7, 2011
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4e3f2684
Valid from: Sun Aug 07 19:57:56 EDT 2011 until: Tue Jul 30 19:57:56 EDT 2041
Certificate fingerprints:
   MD5:  98:84:0E:36:F0:B3:48:9C:CD:13:EB:C6:D8:7F:F3:B1
   SHA1: E6:C5:81:EB:8A:F4:35:B0:04:84:3E:6E:C3:88:BD:B2:66:52:E7:09
   Signature algorithm name: SHA1withRSA
   Version: 3
```

You will need to make note of the `SHA1` entry (see third line from the bottom of the above sample).

Your Google Account

To sign up for an API key, you need a Google account. Ideally, this account would be the same one you intend to use for submitting apps to the Play Store (if, indeed, you intend to do so).

Your API Key

Given that you are logged into the aforementioned Google account, you can visit the Google Cloud Console to request access to the Maps V2 API. They have a tendency to keep changing this set of pages, but these instructions were good as of late February 2014:

- Create a project via the "Create project" option, if you have not done so already for something else (e.g., GCM)
- Open your project, then select "APIs & auth" from the left navigation bar, and in there select "APIs"
- Sift through the various APIs until you find "Google Maps Android API v2", then toggle that on
- Agree to the Terms of Service that appears when you try to toggle on Maps V2 access
- Click "Credentials" in the left navigation bar
- Click the "CREATE NEW KEY" button
- In the popup dialog, choose "Android key"
- In the fields that appear once you chose "Android key", fill in your app's package name and your SHA1 fingerprint, then click the "Create" button

This will give you an "API key" that you will need for your application.

If you wish to have more than one app use Maps V2, you can click "Edit allowed Android applications" for a key, to return to the dialog where you can paste in another SHA1 fingerprint and package name, separated by a semicolon. Or, if you prefer, you can create new keys for each application.

For apps that are in (or going to) production, you will need to supply both the debug and production SHA1 fingerprints with your package name. By doing this on the same key, you will use the same API key string for both debug and production builds, which simplifies things a fair bit over the separate API keys you would have used with the legacy Maps SDK add-on.

Also note that a single API key seems to only support a few fingerprint/package pairs. If you try adding a new pair, and the form ignores you, you will need to set up a separate API key for additional pairs.

The Play Services Library

You also need to set up the Google Play Services library for use with your app.

While this used to be published via a separate "Google Repository" that you had to download to your development machine, nowadays, it is part of Google's overall public Maven repository, the one that `google()` points to in your `build.gradle` files. Most likely, your project is already set up to use this repository.

If so, all you need to do is add a dependency on `com.google.android.gms:play-services-maps` for some likely version (e.g., `com.google.android.gms:play-services-maps:11.6.2`) to your `dependencies` closure.

Note, though, that starting with version 10.x, the `minSdkVersion` imposed by the Play Services libraries is 14. If your desired `minSdkVersion` is lower than that, you will need to remain on older versions of the Play Services libraries.

The Book Samples… And You!

If you wish to try to run the book samples outlined in this chapter, you will need to replace the Maps V2 API key in the manifest with your own.

Setting Up a Basic Map

With that preparation work completed, now you can start working on projects that use the Maps V2 API. In this section, we will review the `MapsV2/Basic` sample project, which simply brings up a Maps V2 map of the world.

The Dependency

Android Studio users need an entry in their top-level `dependencies` closure to pull in the Play Services SDK artifact:

```
dependencies {
    implementation 'com.google.android.gms:play-services-maps:11.6.2'
}
```

(from MapsV2/Basic/app/build.gradle)

The Project Setup and the Manifest

This project uses Maps V2, and so it has a reference to that library project.

Our manifest file is fairly traditional, though there are a number of elements in it that are required by Maps V2:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.mapsv2.basic"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="22" />

  <uses-feature
    android:glEsVersion="0x00020000"
    android:required="false" />

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <meta-data
      android:name="com.google.android.maps.v2.API_KEY"
      android:value="AIzaSyC4iyT46cBOOIdKGcy5EmAxK5uCOQX2Oy8" />

    <meta-data
      android:name="com.google.android.gms.version"
```

```
      android:value="@integer/google_play_services_version" />

  </application>

</manifest>
```

(from MapsV2/Basic/app/src/main/AndroidManifest.xml)

Specifically:

- We need the `WRITE_EXTERNAL_STORAGE` permissions, but only on Android 5.1 and below, so we can use `android:maxSdkVersion="22"` to only request that permission on older devices
- We need a `<meta-data>` element, with a name of `com.google.android.maps.v2.API_KEY`, whose value is the API key we got from the Google APIs Console for use with this particular package name.
- We can have a second `<meta-data>` element, with a name of `com.google.android.gms.version`, with a value of the `@integer/google_play_services_version` (an integer resource supplied by the Play Services SDK library project). Starting with version 8.1.0 of the Maps V2 library, this element is not essential, as it will be added automatically to our manifest via the manifest merger process. However, code written for older versions of Maps V2 than 8.1.0 will need the element, and there is no particular harm in having it.

We also should include a `<uses-feature>` element for OpenGL ES 2.0. If your app absolutely must be able to run Maps V2, have `android:required="true"` (or drop the `android:required` attribute entirely, as `true` is the default), which will force devices to have OpenGL ES 2.0 to run your app. If your app will gracefully degrade for devices incapable of running Maps V2, use `android:required="false"`, as is shown in the sample.

Beyond those items, everything else in this project is based on what the app needs, more so than what Maps V2 needs. Note, though, that the Play Services SDK library project will add additional items to our manifest, notably requests for a few other permissions, like `INTERNET`. Also note that we used to need to define and use a custom permission, based upon our app's package name and ending in `MAPS_RECEIVE`. This is not required as of Play Services 3.1.59 and the "rev 8" release of the Play Services SDK.

The Play Services Detection

In the fullness of time, all devices that are capable of using Maps V2 will already have the on-device portion of this logic, known as the "Google Play services" app.

However, it is entirely possible, in the short term, that you will encounter devices that are capable of using Maps V2 (e.g., they have OpenGL ES 2.0 or higher), but do not have the "Google Play services" app from the Play Store, and therefore you cannot actually *use* Maps V2 in your app.

This is a departure from the Maps V1 approach, where either the device shipped with maps capability, or it did not, and nothing (legally) could be done to change that state.

To determine whether or not the Maps V2 API is available to you, the best option is to call the `isGooglePlayServicesAvailable()` static method on the `GooglePlayServicesUtil` utility class supplied by the Play Services library. This will return an `int`, with a value of `ConnectionResult.SUCCESS` if Maps V2 can be used right away.

Actually assisting the user to get Maps V2 set up all the way is conceivable but is also bug-riddled and annoying. The `MapsV2/Basic` sample app has an `AbstractMapActivity` base class that is designed to hide most of this annoyance from you. If you wish to know the details of how this works, we will cover it later in this chapter.

The Fragment and Activity

Our main activity — `MainActivity` — extends from the aforementioned `AbstractMapActivity` and simply overrides `onCreate()`, as most activities do:

```
package com.commonsware.android.mapsv2.basic;

import android.os.Bundle;

public class MainActivity extends AbstractMapActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
      setContentView(R.layout.activity_main);
```

```
    }
  }
}
```

(from MapsV2/Basic/app/src/main/java/com/commonsware/android/mapsv2/basic/MainActivity.java)

We call `setContentView()` to load up the `activity_main` layout resource:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/map"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  class="com.google.android.gms.maps.MapFragment"/>
```

(from MapsV2/Basic/app/src/main/res/layout/activity_main.xml)

That resource, in turn, has a `<fragment>` element pointing to a `com.google.android.gms.maps.MapFragment` class supplied by the Play Services library. This is a fragment that knows how to display a Maps V2 map. There is a corresponding `com.google.android.gms.maps.SupportMapFragment` class for use with the fragments backport from the Android Support library.

You will notice, though, that we only call `setContentView()` if a `readyToGo()` method returns `true`. The `readyToGo()` method is supplied by the `AbstractMapActivity` class and returns `true` if we are safe to go ahead and use Maps V2, `false` otherwise. In the `false` case, `AbstractMapActivity` will be taking care of trying to get Maps V2 going, and we need do nothing further.

The Result

When you run the app, assuming that Maps V2 is ready for use, you will get a basic map showing a good-sized chunk of the planet:

Figure 595: Maps V2 Map, as Initially Viewed

If your Maps V2 API key is incorrect, or you do not have this app's package name set up for that key in the Google APIs Console, you will get an "Authorization failure" error message in Logcat, and you will get a blank map, akin to the behavior seen in Maps V1 when you had an invalid `android:apiKey` attribute on the `MapView`.

Playing with the Map

Showing a map of a good-sized chunk of the planet is nice, and it is entirely possible that is precisely what you wanted to show the user. If, on the other hand, you wanted to show the user something else — another location, a closer look, etc. — you will need to further configure your map, via a `GoogleMap` object.

To see how this is done, take a look at the `MapsV2/NooYawk` sample application. This is a clone of `MapsV2/Basic` that adds in logic to center and zoom the map over a portion of New York City.

The `onCreate()` method of the revised `MapActivity` is now a bit more involved:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
      setContentView(R.layout.activity_main);

      MapFragment mapFrag=
          (MapFragment)getFragmentManager().findFragmentById(R.id.map);

      if (savedInstanceState == null) {
        mapFrag.getMapAsync(this);
      }
    }
  }
```

(from MapsV2/NooYawk/app/src/main/java/com/commonsware/android/mapsv2/nooyawk/MainActivity.java)

After calling `setContentView()`, we can retrieve our `MapFragment` via `findFragmentById()`, no different than any other static fragment.

Then, if `savedInstanceState` is `null` — meaning that the activity is not being recreated, but instead is being created from scratch — we call `getMapAsync()` on the `MapFragment`. This triggers some asynchronous work to set up a `GoogleMap` object. `getMapAsync()` takes an implementation of `OnMapReadyCallback` as a parameter. In this case, `OnMapReadyCallback` is implemented on the activity itself.

That `GoogleMap` object will then be delivered to us in `onMapReady()`. Most of our work in configuring the map will be accomplished by calling methods on this `GoogleMap` object:

```
  @Override
  public void onMapReady(GoogleMap map) {
    CameraUpdate center=
        CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044,
            -73.98180484771729));
    CameraUpdate zoom=CameraUpdateFactory.zoomTo(15);

    map.moveCamera(center);
    map.animateCamera(zoom);
  }
```

(from MapsV2/NooYawk/app/src/main/java/com/commonsware/android/mapsv2/nooyawk/MainActivity.java)

To change where the map is centered, we can create a `CameraUpdate` object from the `CameraUpdateFactory` ("camera" in this case referring to the position of the user's virtual eyes with respect to the surface of the Earth). The `newLatLng()` factory

method on `CameraUpdateFactory` will give us a `CameraUpdate` object that can re-center the map over a supplied latitude and longitude. Those coordinates are encapsulated in a `LatLng` object and are maintained as decimal degrees as Java `float` or `double` values (as opposed to the Maps V1 `GeoPoint`, which used integer microdegrees).

To change the zoom level of the map, we need another `CameraUpdate` object, this time from the `zoomTo()` factory method on `CameraUpdateFactory`. As with Maps V1, the zoom levels start at 1 and zoom in by powers of two. As you will see, a value of 15 gives you a nice block-level view of a city like New York City.

To actually apply these changes to the map, we have two methods on `GoogleMap`:

1. `moveCamera()` will perform a "smash cut" and immediately change the map based upon the supplied `CameraUpdate`
2. `animateCamera()` will smoothly animate the map from its original state to the new state supplied by the `CameraUpdate`

In our case, we immediately shift to the proper position, but then zoom in from the default zoom level to 15, giving us a map centered over Columbus Circle, in the southwest corner of Central Park in Manhattan:

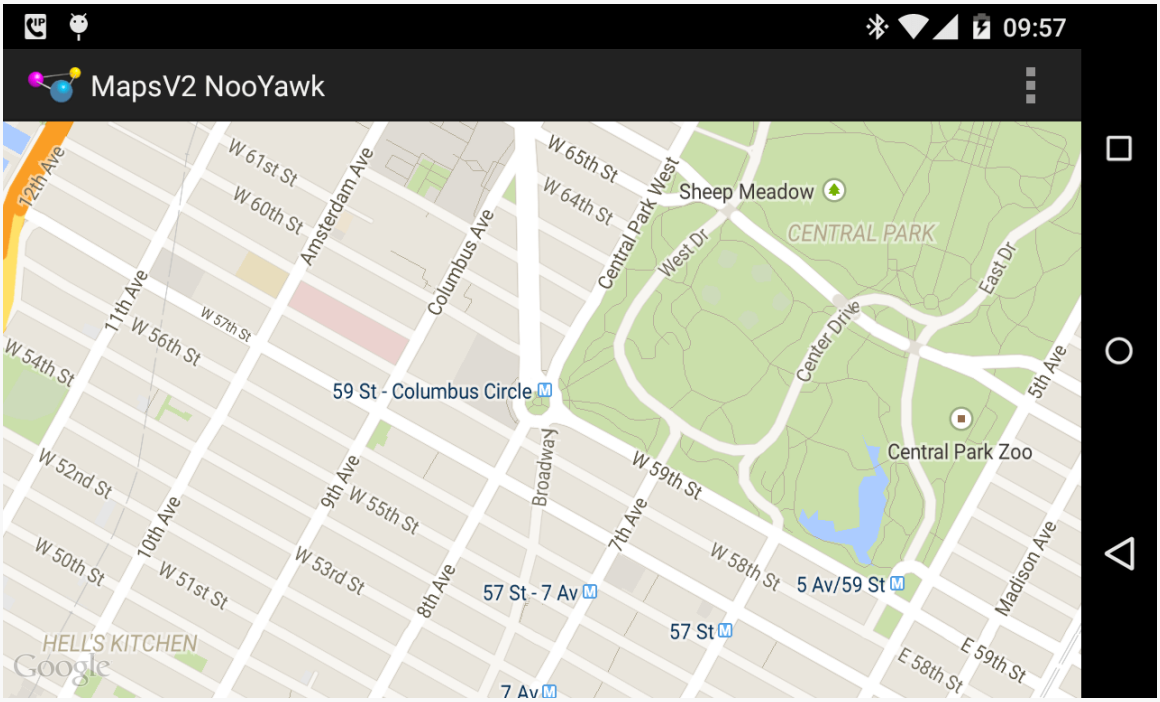


Figure 596: Maps V2 Centered Over Columbus Circle, New York City

Note that you might want to do both actions simultaneously, rather than have one be animated and one not as in this sample. In that case, you can manually create a `CameraPosition` object that describes the desired center, zoom, etc., then use the `newCameraPosition()` method on `CameraUpdateFactory` to get a `CameraUpdate` instance that will apply all of those changes.

Map Tiles

The map, by default, shows the normal tile set. `setMapType()` on the `GoogleMap` allows you to switch to satellite, hybrid (satellite view plus place labels), or terrain tile sets.

Placing Simple Markers

For markers — push-pins and the like — you simply hand markers to the `GoogleMap` for display, as is illustrated in the `MapsV2/Markers` sample application. This is a clone of `MapsV2/NooYawk`, with four markers for four landmarks within Manhattan.

Our `onCreate()` method on `MainActivity` now always invokes `getMapAsync()`, not just when the activity is first created. However, we still check `savedInstanceState` and set a new `needsInit` `boolean` data member to `true` if `savedInstanceState` is `null`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (readyToGo()) {
    setContentView(R.layout.activity_main);

    MapFragment mapFrag=
        (MapFragment)getFragmentManager().findFragmentById(R.id.map);

    if (savedInstanceState == null) {
      needsInit=true;
    }

    mapFrag.getMapAsync(this);
  }
}
```

(from MapsV2/Markers/app/src/main/java/com/commonsware/android/mapsv2/markers/MainActivity.java)

Our `onMapReady()` method performs the camera adjustments if `needsInit` is `true`. It also has four additional statements – calls to a private `addMarker()` method to define the four landmarks:

```
@Override
public void onMapReady(GoogleMap map) {
  if (needsInit) {
    CameraUpdate center=
        CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044,
                                                 -73.98180484771729));
    CameraUpdate zoom=CameraUpdateFactory.zoomTo(15);

    map.moveCamera(center);
    map.animateCamera(zoom);
  }

  addMarker(map, 40.748963847316034, -73.96807193756104,
            R.string.un, R.string.united_nations);
  addMarker(map, 40.76866299974387, -73.98268461227417,
      R.string.lincoln_center,
      R.string.lincoln_center_snippet);
  addMarker(map, 40.765136435316755, -73.97989511489868,
```

```
        R.string.carnegie_hall, R.string.practice_x3);
    addMarker(map, 40.70686417491799, -74.01572942733765,
        R.string.downtown_club, R.string.heisman_trophy);
}
```

(from MapsV2/Markers/app/src/main/java/com/commonsware/android/mapsv2/markers/MainActivity.java)

The `addMarker()` method on our `MainActivity` adds markers by creating a `MarkerOptions` object and passing it to the `addMarker()` on `GoogleMap`. `MarkerOptions` offers a so-called "fluent" interface, with a series of methods to affect one aspect of the `MarkerOptions`, each of which returns the `MarkerOptions` object itself. That way, configuring a `MarkerOptions` is a chained series of method calls:

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet) {
  map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                   .title(getString(title))
                                   .snippet(getString(snippet)));
}
```

(from MapsV2/Markers/app/src/main/java/com/commonsware/android/mapsv2/markers/MainActivity.java)

Here, we:

- Set the `position()` of the marker, in the form of another `LatLng` object
- Set the `title()` and `snippet()` of the marker to be a pair of strings loaded from string resources

We will see other methods available on `MarkerOptions` in upcoming sections of this chapter.

`addMarker()` on `GoogleMap` returns an actual `Marker` object, which we could hold onto to change certain aspects of it later on (e.g., its title). In the case of this sample, we ignore this.

Now, you may be wondering why we set up the markers on every `onMapReady()` invocation, not just in the `needsInit` block. That is because while a `MapFragment` retains its camera information (center, zoom, etc.) on a configuration change, it does *not* retain its markers. Hence, we need to re-establish the markers in all calls to `onCreate()`, not just the very first one.

With no other changes, we get a version of the map that shows markers at our designated locations:

Figure 597: Maps V2 with Two Markers

Initially, we only see two markers, as the other two are outside the current center position and zoom level of the map. If the user changes the center or zoom, markers will come and go as needed:

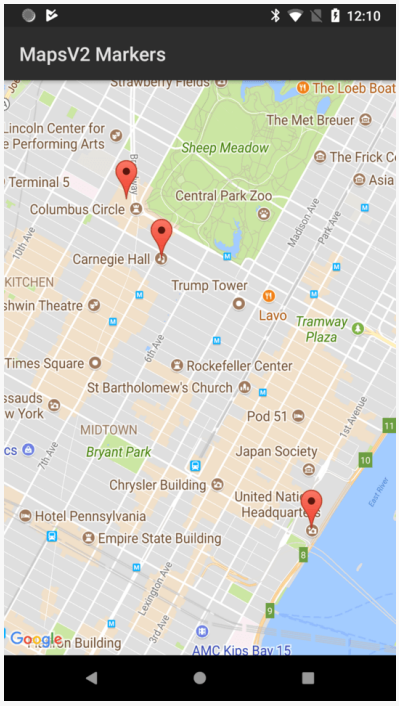


Figure 598: Maps V2 with All Four Markers

We do not need to worry about managing the markers ourselves, so long as the `GoogleMap` performance is adequate. It is likely that dumping 10,000 markers into a `GoogleMap` will still result in sluggish responses, though, so you may need to add and remove markers yourself based upon what portion of the world the user happens to be examining in the map at the moment.

Seeing All the Markers

When you add markers to a map, there is no guarantee that the markers will be visible given the map's current center position and zoom level. In fact, it is entirely possible that you add a bunch of markers and *none* are visible, so the user may not realize that the markers were added.

There is a way that you can center and zoom the map to show some set of markers, based on their positions. You get to choose the markers: all of them, the four nearest markers, etc.

We can see how this works in the MapsV2/Bounds sample application. This is a clone of MapsV2/Markers from the previous section, with reworked code to show all four markers when the map is first displayed.

The key to making this work is a LatLngBounds object. This represents a bounding box that contains all LatLng locations handed to the LatLngBounds. To build up a LatLngBounds, you can use the LatLngBounds.Builder class. So, our revised MainActivity has a LatLngBounds.Builder private data member:

```
private LatLngBounds.Builder builder=new LatLngBounds.Builder();
```

(from MapsV2/Bounds/app/src/main/java/com/commonsware/android/mapsv2/markers/MainActivity.java)

Our revised addMarker() method adds the LatLng values from our markers as they are added to the map:

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet) {
  Marker marker=
      map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                       .title(getString(title))
                                       .snippet(getString(snippet)));

  builder.include(marker.getPosition());
}
```

(from MapsV2/Bounds/app/src/main/java/com/commonsware/android/mapsv2/markers/MainActivity.java)

Finally, the revised onMapReady() moves the CameraUpdateFactory work until after all four of the addMarker() calls and changes it a bit:

```
@Override
public void onMapReady(final GoogleMap map) {
  addMarker(map, 40.748963847316034, -73.96807193756104,
            R.string.un, R.string.united_nations);
  addMarker(map, 40.76866299974387, -73.98268461227417,
            R.string.lincoln_center,
            R.string.lincoln_center_snippet);
  addMarker(map, 40.765136435316755, -73.97989511489868,
            R.string.carnegie_hall, R.string.practice_x3);
  addMarker(map, 40.70686417491799, -74.01572942733765,
            R.string.downtown_club, R.string.heisman_trophy);

  if (needsInit) {
    findViewById(android.R.id.content).post(new Runnable() {
      @Override
```

```
      public void run() {
        CameraUpdate allTheThings=
            CameraUpdateFactory.newLatLngBounds(builder.build(), 32);

        map.moveCamera(allTheThings);
      }
    });
  }
}
```

(from MapsV2/Bounds/app/src/main/java/com/commonsware/android/mapsv2/markers/MainActivity.java)

Specifically, we:

- Ask the `LatLngBounds.Builder` to `build()` the `LatLngBounds`
- Pass that to the `newLatLngBounds()` method on `CameraUpdateFactory`, along with an inset value in pixels (all `LatLng` locations will be that many pixels in from the edges, or more)
- Use `moveCamera()` to center and zoom the map based upon the resulting `CameraUpdate`

All of this is done in a `Runnable` which we `post()` to a `View` (here, the `FrameLayout` of our activity supplied by Android as `android.R.id.content`). `GoogleMap` cannot ensure that all of our markers are visible until it knows how big the map is, and that is not known until the map is rendered to the screen. `post()` will add our work to the end of the main application thread's work queue. The `Runnable` will not be run until after the map is on the screen, at which time the `CameraUpdate` can work.

Flattening and Rotating Markers

Markers, by default, appear to be "push pins" pressed into the surface of the map. This is not necessarily obvious with the default top-down perspective of the map camera. But, if you use a two-finger vertical swiping gesture, you can change the camera tilt, and that will illustrate the "push pin" effect a bit better:

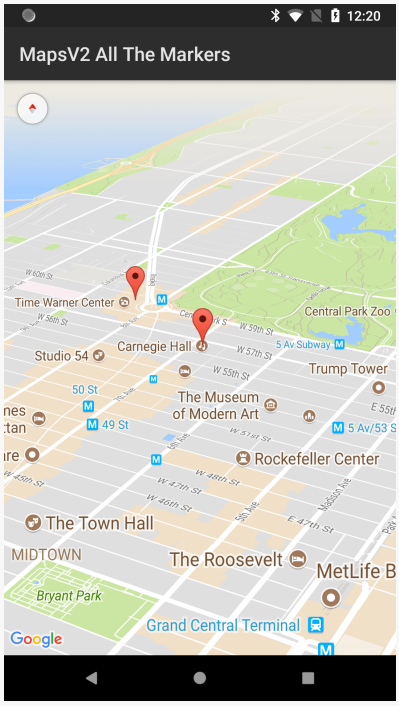


Figure 599: Maps V2 with Markers, Viewed on a Tilt

However, you have options for flat markers and rotated markers.

A flat marker is one that is flat on the map. In other words, rather than theoretically rising out of the Z axis of the map, the marker is kept on the X-Y plane:

Figure 600: Maps V2 with Markers, One Normal, One Flat

It is also possible to rotate a marker. The flat marker in the previous screenshot is rotated 90 degrees from its normal "bulb on the north side" orientation. The following screenshot shows another flat marker, rotated 270 degrees from normal:

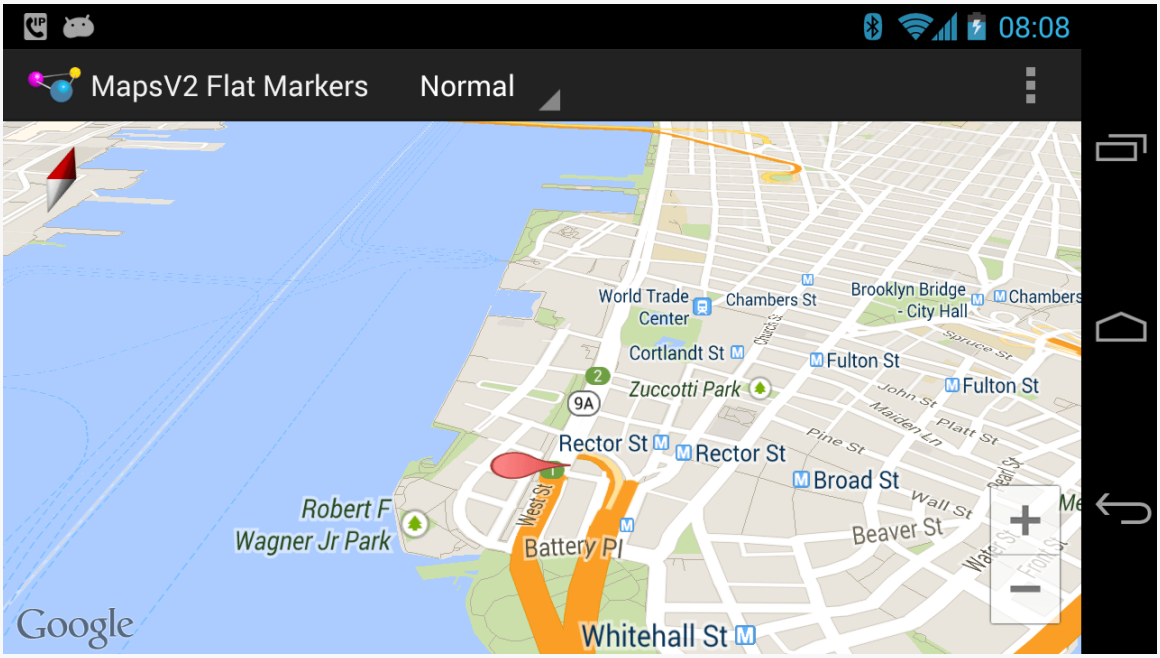


Figure 601: Maps V2 with Markers, Flat and Rotated

These features can be handy for providing pointers in a particular direction, such as indicating not only the location to make a turn, but what direction to turn at that location.

These capabilities are courtesy of `flat()` and `rotation()` methods on `MarkerOptions`, plus corresponding getters and setters on `Marker` itself. To see how this works, let's examine the `MapsV2/FlatMarkers` sample application. This is a clone of `MapsV2/Markers`, with markers applied using different values for `flat()` and `rotation()`.

Specifically, our own `addMarker()` helper method now takes and applies a `boolean` parameter for `flat` (`true` means it is flat, `false` means normal behavior), as well as a `float` parameter for `rotation` (a value between `0` and `360` for the rotation off the default in degrees):

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet, boolean flat,
                       float rotation) {
  map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                   .title(getString(title))
                                   .snippet(getString(snippet))
```

```
                                    .flat(flat).rotation(rotation));
  }
```

(from MapsV2/FlatMarkers/app/src/main/java/com/commonsware/android/mapsv2/flatmarkers/MainActivity.java)

When we call `addMarker()`, we supply corresponding values:

```
    addMarker(map, 40.748963847316034, -73.96807193756104,
              R.string.un, R.string.united_nations, false, 180);
    addMarker(map, 40.76866299974387, -73.98268461227417,
              R.string.lincoln_center,
              R.string.lincoln_center_snippet, false, 0);
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3, true, 90);
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy, true,
              270);
```

(from MapsV2/FlatMarkers/app/src/main/java/com/commonsware/android/mapsv2/flatmarkers/MainActivity.java)

Sprucing Up Your “Info Windows”

If the user taps on one of the markers from the preceding sample, Android will automatically display a popup, known as an “info window”:

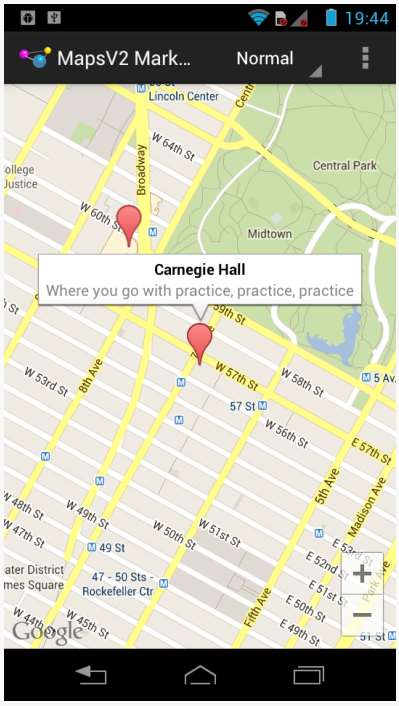


Figure 602: Maps V2 with Default Info Window

You can tailor that "info window" if desired, either replacing just the interior portion (leaving the bounding border with its caret intact) or replacing the entire window. However, in the interests of memory conservation, you do not hand new `View` widgets to the `MarkerOptions` object. Instead, you can provide an adapter that will be called when info windows (or their contents) are required.

To see how this works, we can examine the `MapsV2/Popups` sample application. This is a clone of `MapsV2/Markers`, where we are using our own layout file for the contents of the info windows, from the `popup.xml` layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:padding="2dip"
```

```
    android:src="@drawable/ic_launcher"
    android:contentDescription="@string/icon"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/snippet"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

(from MapsV2/Popups/app/src/main/res/layout/popup.xml)

Here, we will show the title and snippet in our own chosen font size and weight, plus show the launcher icon on the left.

To use this layout, we must create an `InfoWindowAdapter` implementation — in the case of this sample project, that is found in the `PopupAdapter` class:

```
package com.commonsware.android.mapsv2.popups;

import android.annotation.SuppressLint;
import android.view.LayoutInflater;
import android.view.View;
import android.widget.TextView;
import com.google.android.gms.maps.GoogleMap.InfoWindowAdapter;
import com.google.android.gms.maps.model.Marker;

class PopupAdapter implements InfoWindowAdapter {
  private View popup=null;
  private LayoutInflater inflater=null;

  PopupAdapter(LayoutInflater inflater) {
    this.inflater=inflater;
```

```
  }

  @Override
  public View getInfoWindow(Marker marker) {
    return(null);
  }

  @SuppressLint("InflateParams")
  @Override
  public View getInfoContents(Marker marker) {
    if (popup == null) {
      popup=inflater.inflate(R.layout.popup, null);
    }

    TextView tv=(TextView)popup.findViewById(R.id.title);

    tv.setText(marker.getTitle());
    tv=(TextView)popup.findViewById(R.id.snippet);
    tv.setText(marker.getSnippet());

    return(popup);
  }
}
```

(from MapsV2/Popups/app/src/main/java/com/commonsware/android/mapsv2/popups/PopupAdapter.java)

When an info window is to be displayed, Android will first call `getInfoWindow()` on our `InfoWindowAdapter`, passing in the `Marker` whose info window is needed. If we return a `View` here, that will be used for the entire info window. If, instead, we return `null`, Android will call `getInfoContents()`, passing in the same `Marker` object. If we return a `View` here, Android will use that as the "body" of the info window, with Android supplying the border. If we return `null`, the default info window is displayed. This way, we can conditionally do any of the three possibilities (replace the window, replace the contents, or accept the default).

In our case, `getInfoContents()` will inflate the `popup.xml` layout and populate the two `TextView` widgets with the title and snippet from the `Marker`. However, we cache the inflated layout and reuse it on the second and subsequent calls to `getInfoContents()`. Despite the "adapter" name conjuring up visions of `ListAdapter` and having multiple outstanding views, `InfoWindowAdapter` will only ever use one `View` at a time. Hence, rather than inflate our layout each time we need to show the info window, we can safely reuse the previously-used `View`.

Then, we just need to tell the `GoogleMap` to use our `InfoWindowAdapter`, via a call to `setInfoWindowAdapter()`, such as this statement from `onMapReady()` of our new edition of `MainActivity`:

```
    map.setInfoWindowAdapter(new PopupAdapter(getLayoutInflater()));
```

(from MapsV2/Popups/app/src/main/java/com/commonsware/android/mapsv2/popups/MainActivity.java)

Now, when the user taps on a marker, they will get our customized info window:

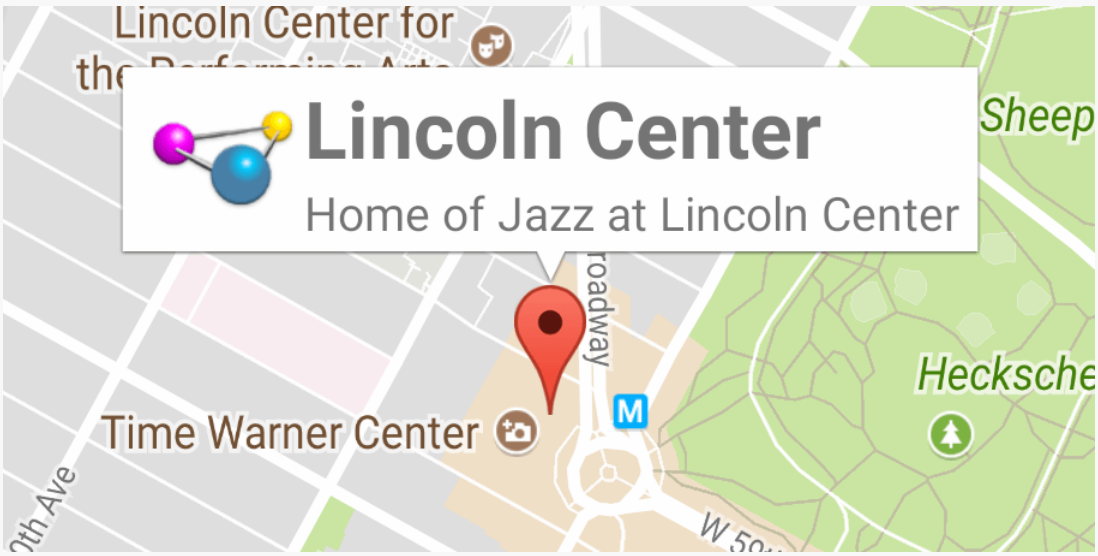


Figure 603: Maps V2 with Customized Info Window

We can also call `setOnInfoWindowClickListener()` on our `GoogleMap`, passing in an implementation of the `OnInfoWindowClickListener` interface, to find out when the user taps on the info window. In the case of `MainActivity`, we set up the activity itself to implement that interface and be the listener:

```
    map.setOnInfoWindowClickListener(this);
```

(from MapsV2/Popups/app/src/main/java/com/commonsware/android/mapsv2/popups/MainActivity.java)

This requires us to implement an `onInfoWindowClick()` method, where we are passed the `Marker` representing the tapped-upon info window:

```
@Override
public void onInfoWindowClick(Marker marker) {
  Toast.makeText(this, marker.getTitle(), Toast.LENGTH_LONG).show();
}
```

(from MapsV2/Popups/app/src/main/java/com/commonsware/android/mapsv2/popups/MainActivity.java)

Here, we just display a `Toast` with the title of the `Marker` when the user taps an info window:

Figure 604: Maps V2 with Toast Triggered by Tap on Info Window

Note that, according to the documentation, you can only find out about taps on the entire info window. Indeed, if you try setting up click listeners on the widgets in your custom layout, you will find that they are not called. This is because the `View` you return for the info window is converted into a `Bitmap`, which is then displayed. Presumably, this is to steer developers in the direction of making larger tap targets, rather than expecting users to tap tiny elements within an info window. On the other hand, if your design calls for a large info window containing several navigation options, you will need to either re-think your design or avoid the info window system. We will see how to find out about taps on markers more directly later in this chapter.

Images and Your Info Window

The `Bitmap` approach that Maps V2 uses for the info window introduces an additional challenge: updating the info window itself. Normally, we would just

update the individual widgets in the info window, the way we might update widgets in an already-visible row in a `ListView`. However, that is not an option here, as our widgets are discarded almost immediately.

One particular occurrence of this problem comes when you want to show an image in the info window. If the image is a resource, or is already in memory, showing it is not a big problem, as you can just populate your `ImageView` in your info window with it. However, if the image is a file (or, worse, needs to be downloaded), you want to load the image asynchronously. However, if you kick off some background thread, like an `AsyncTask`, to retrieve the image, you will return from your `InfoWindowAdapter` method long before the task is complete. Your info window will show whatever placeholder image you used; the image you loaded will never be seen, even if you update your original `ImageView`.

There are two solutions to this problem.

The best solution, by far, is to have the images before you need them, wherever possible. For example, if you are showing a map with 25 markers, for which you need 25 thumbnail images, start downloading those images while you are showing the map. With luck, at the point in time when the user taps on a marker to show the info window, you will have your image already.

However, this approach will not work well if:

- You need a ridiculous number of images, or
- You need images, but they need to be downloaded full-sized and turned into thumbnails locally, as that might consume quite a bit of bandwidth, or
- Your last name is Murphy, and therefore the user taps on an info window before you have had a chance to prepare its image

The workaround is to make note of the `Marker` the user tapped upon to open its info window, then call `showInfoWindow()` on that `Marker` to cause the info window to be redisplayed once you have your image, triggering calls to your `InfoWindowAdapter`. There, you can see that your image cache includes the image that you need, and you can apply it to the info window.

The problem *here* is that it is possible that the user tapped on another marker, after the first one, while you were busily fetching and loading the image. Hence, rather than blindly calling `showInfoWindow()` on the `Marker`, you should call `isInfoWindowShown()` first, and only call `showInfoWindow()` to force the refresh if `isInfoWindowShown()` returns `true`. Otherwise, some other marker's info window is

shown. The user is not expecting this earlier info window to somehow magically reappear.

All of this is a pain. It can be made a bit less of a pain by use of an image fetching-and-caching library like Picasso. We can see how this can be applied by looking at the MapsV2/ImagePopups sample application. This is a clone of MapsV2/Popups, with some additions to handle lazy-populating an info window based upon a downloaded image.

First, since we are going to be generating some thumbnails based on downloaded imagery, it helps to establish a fixed-size ImageView for our icon. So, this project has a pair of dimension resources, for the image height and width:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="icon_width">96dp</dimen>
    <dimen name="icon_height">64dp</dimen>

</resources>
```

(from MapsV2/ImagePopups/app/src/main/res/values/dimens.xml)

Those are then used in a revised version of the popup layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="@dimen/icon_width"
    android:layout_height="@dimen/icon_height"
    android:padding="2dip"
    android:src="@drawable/ic_launcher"
    android:contentDescription="@string/icon"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:orientation="vertical">

    <TextView
      android:id="@+id/title"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/snippet"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

(from MapsV2/ImagePopups/app/src/main/res/layout/popup.xml)

We need some way of keeping track of what images should be used for each marker. This is somewhat annoying to implement, as we cannot subclass `Marker`, since it is marked as `final` and cannot be extended. However, we can use `getId()` on a `Marker` to obtain a unique ID, and we can use that as the key to additional model data. We will examine variations on this technique later in this chapter. For now, this sample gets away with a simple `HashMap`, mapping the string ID of a `Marker` to a `Uri` representing an image to be shown for that `Marker`'s info window:

```
  private HashMap<String, Uri> images=new HashMap<String, Uri>();
```

(from MapsV2/ImagePopups/app/src/main/java/com/commonsware/android/mapsv2/imagepopups/MainActivity.java)

Our private `addMarker()` method now takes a `String` name of an image, and it adds a `Uri` pointing to that image to the `HashMap`, keyed by the ID of the generated `Marker`:

```
  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet, String image) {
    Marker marker=
        map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                         .title(getString(title))
                                         .snippet(getString(snippet)));

    if (image != null) {
      images.put(marker.getId(),
                 Uri.parse("http://misc.commonsware.com/mapsv2/"
                     + image));
    }
  }
```

(from MapsV2/ImagePopups/app/src/main/java/com/commonsware/android/mapsv2/imagepopups/MainActivity.java)

For three of our markers, we pass in actual filenames; for a fourth, `null` is used, indicating that there is no suitable image for use:

```
    addMarker(map, 40.748963847316034, -73.96807193756104,
              R.string.un, R.string.united_nations, "UN_HQ.jpg");
    addMarker(map, 40.76866299974387, -73.98268461227417,
              R.string.lincoln_center,
              R.string.lincoln_center_snippet,
              "Avery_Fisher_Hall.jpg");
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3,
              "Carnegie_Hall.jpg");
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy, null);
```

(from MapsV2/ImagePopups/app/src/main/java/com/commonsware/android/mapsv2/imagepopups/MainActivity.java)

Note that the three images being used in this chapter come from Wikipedia. One is public domain, the others are licensed under the Creative Commons Attribution 1.0 license. Those two are a picture of Avery Fisher Hall, part of the Lincoln Center for the Performing Arts (courtesy of Geographer) and the other is a picture of the United Nations building (courtesy of WorldIslandInfo).

The `PopupAdapter` needs access to these images. It will also need access to a `Context`, for use with Picasso. So, `PopupAdapter` now has data members for these, which are passed into a revised version of its constructor by `MainActivity`. That constructor not only holds onto the new objects, but it retrieves the values of the dimension resources for our images, converted by Android into pixels for the screen density of the device that we are running on:

```
PopupAdapter(Context ctxt, LayoutInflater inflater,
             HashMap<String, Uri> images) {
  this.ctxt=ctxt;
  this.inflater=inflater;
  this.images=images;

  iconWidth=
      ctxt.getResources().getDimensionPixelSize(R.dimen.icon_width);
  iconHeight=
      ctxt.getResources().getDimensionPixelSize(R.dimen.icon_height);
}
```

(from MapsV2/ImagePopups/app/src/main/java/com/commonsware/android/mapsv2/imagepopups/PopupAdapter.java)

The revised `getInfoContents()` method is significantly more complicated than was its predecessor:

```
@SuppressLint("InflateParams")
@Override
public View getInfoContents(Marker marker) {
  if (popup == null) {
    popup=inflater.inflate(R.layout.popup, null);
  }

  if (lastMarker == null
      || !lastMarker.getId().equals(marker.getId())) {
    lastMarker=marker;

    TextView tv=(TextView)popup.findViewById(R.id.title);

    tv.setText(marker.getTitle());
    tv=(TextView)popup.findViewById(R.id.snippet);
    tv.setText(marker.getSnippet());

    Uri image=images.get(marker.getId());
    ImageView icon=(ImageView)popup.findViewById(R.id.icon);

    if (image == null) {
      icon.setVisibility(View.GONE);
    }
    else {
      Picasso.with(ctxt).load(image).resize(iconWidth, iconHeight)
             .centerCrop().noFade()
             .placeholder(R.drawable.placeholder)
             .into(icon, new MarkerCallback(marker));
    }
  }

  return(popup);
}
```

(from MapsV2/ImagePopups/app/src/main/java/com/commonsware/android/mapsv2/imagepopups/PopupAdapter.java)

We track the last `Marker` that we have processed in a `lastMarker` data member. Initially, of course, that will be `null`. If it is, or if the `Marker` passed into `getInfoContents()` is a different one (based on the `getId()` value), then we populate the popup `View`. This includes fetching the `Uri` from the `HashMap` of `Uri` values (given the `Marker` ID). If there is no `Uri`, `getInfoContents()` marks the `ImageView` as `GONE`, so it will not take up space in the popup. If, however, there *is* an image `Uri`, `getInfoContents()` asks Picasso to "do its thing":

- Load the image from the `Uri`
- Resize the image to be the desired dimensions for the `ImageView`, center-cropping to keep the right aspect ratio
- Skip the fade-in animation that is normally applied when Picasso populates an `ImageView` (as the Maps V2 `Bitmap` is generated before the animation completes, resulting in a washed-out image)
- Use a particular `placeholder` drawable resource while the image is loading
- Populate the `ImageView` with the results, specifying a `MarkerCallback` to be notified of the results

`MarkerCallback`, as an implementation of Picasso's `Callback` interface, needs `onError()` and `onSuccess()` methods. `onError()` just dumps a message to Logcat, while `onSuccess()` refreshes the info window, via a call to `showInfoWindow()` on the `Marker`, if that info window is still showing:

```
static class MarkerCallback implements Callback {
  Marker marker=null;

  MarkerCallback(Marker marker) {
    this.marker=marker;
  }

  @Override
  public void onError() {
    Log.e(getClass().getSimpleName(), "Error loading thumbnail!");
  }

  @Override
  public void onSuccess() {
    if (marker != null && marker.isInfoWindowShown()) {
      marker.showInfoWindow();
    }
  }
}
```

(from MapsV2/ImagePopups/app/src/main/java/com/commonsware/android/mapsv2/imagepopups/PopupAdapter.java)

If you run this sample app, you will see the popup with a placeholder image at first, quickly being replaced by the thumbnail supplied by Picasso:

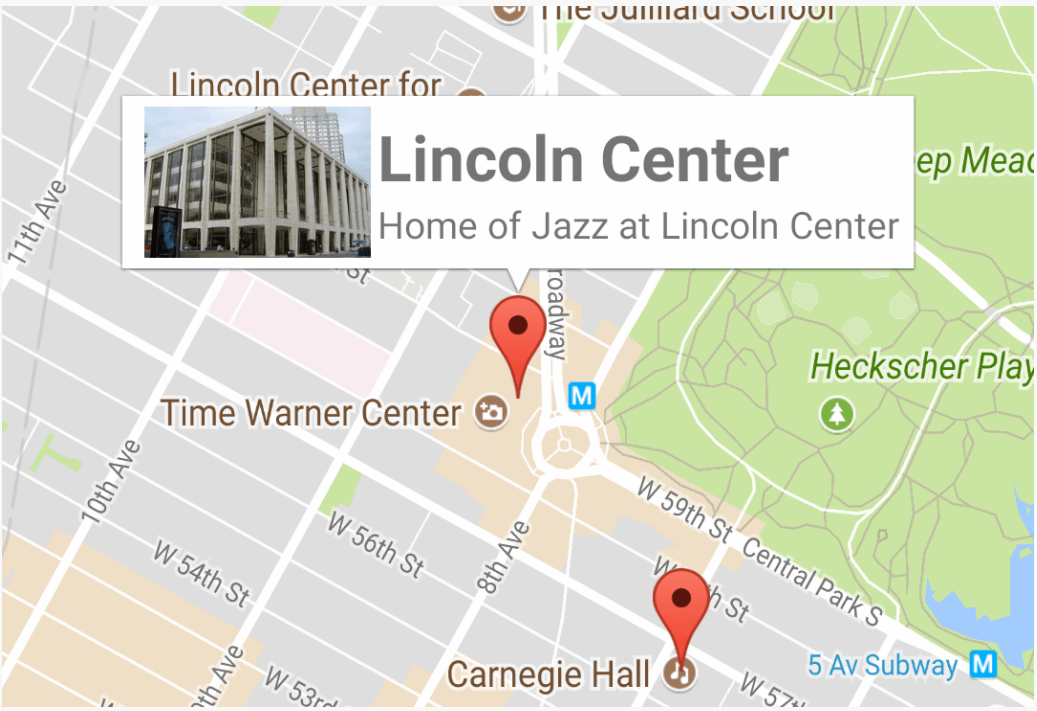


Figure 605: Maps V2 with Popup and Thumbnail

Setting the Marker Icon

Maps V2 includes a stock marker icon that looks a lot like the standard Google Maps marker. You have three major choices for what to use for your own markers:

1. Stick with the stock icon, which is the default behavior
2. Change the stock icon to a different hue
3. Replace the stock icon with your own from an asset, resource, file, or in-memory `Bitmap`

To indicate that you want a different icon than the stock one, use the `icon()` method on the `MarkerOptions` fluent interface. This takes a `BitmapDescriptor`, which you get from one of a series of static methods on the `BitmapDescriptorFactory` class.

For example, you might have a revised version of the `addMarker()` method of `MainActivity` that took a hue — a value from 0 to 360 representing different colors along a color wheel. 0 represents red, 120 represents green, and 240 represents blue, with different shades in between. There is a series of `HUE_` constants defined on

`BitmapDescriptorFactory`, plus a `defaultMarker()` method that takes a hue as a parameter and returns a `BitmapDescriptor` that will use the stock icon, colored to the specified hue:

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet, int hue) {
  map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                   .title(getString(title))
                                   .snippet(getString(snippet))
                                   .icon(BitmapDescriptorFactory.defaultMarker(hue)));
}
```

(from MapsV2/Taps/app/src/main/java/com/commonsware/android/mapsv2/taps/MainActivity.java)

This could then be used to give you different colors per marker, or by category of marker, etc.:

Figure 606: Maps V2 with Alternate Marker Hues

Note that you can modify the icon at runtime via the `setIcon()` method on the `Marker` returned by `addMarker()` method on `GoogleMap`.

However, you cannot draw the marker directly yourself, the way you might have with Maps V1. What you *can* do is draw to a `Bitmap`-backed `Canvas` object, then use the

resulting `Bitmap` with `BitmapFactoryDescriptor` and its `fromBitmap()` factory method.

Responding to Taps

Perhaps we would like to find out when a user taps on one of our markers, instead of displaying an info window. Maybe we want to have some other UI response to that tap in our app.

To do that, simply create an implementation of the `OnMarkerClickListener` interface and attach it to the `GoogleMap` via `setOnMarkerClickListener()`. You will then be called with `onMarkerClick()` when the user taps on a marker, and you are passed the `Marker` object in question. If you return `true`, you are indicating that you are handling the event; returning `false` means that default handling (the info window) should be done.

You can see this, plus the multi-colored markers, in the `MapsV2/Taps` sample application. This takes `MapsV2/Popups` and adds a `Toast` when the user taps a marker, in addition to displaying the info window:

```
@Override
public boolean onMarkerClick(Marker marker) {
  Toast.makeText(this, marker.getTitle(), Toast.LENGTH_LONG).show();

  return(false);
}
```

(from MapsV2/Taps/app/src/main/java/com/commonsware/android/mapsv2/taps/MainActivity.java)

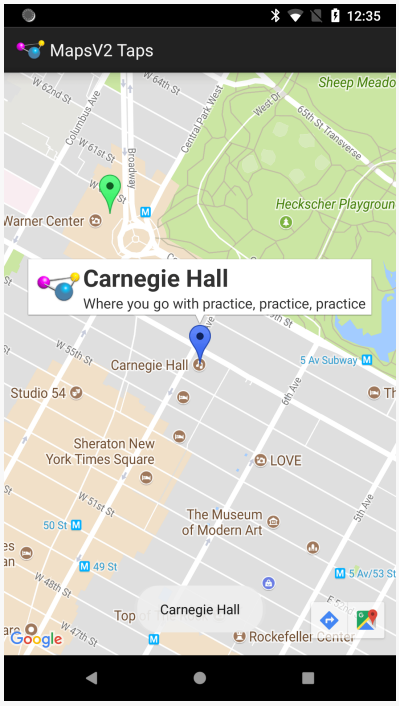


Figure 607: Maps V2 with Toast and Info Window

Our call to `setOnMarkerClickListener()` is up in the `onMapReady()` method of `MainActivity`:

```
    map.setOnMarkerClickListener(this);
```

(from MapsV2/Taps/app/src/main/java/com/commonsware/android/mapsv2/taps/MainActivity.java)

Dragging Markers

By default, markers are not draggable. But, if you call `draggable(true)` on your `MarkerOptions` when creating the marker — or call `setDraggable(true)` on the `Marker` later on — Android will automatically support drag-and-drop. The user can tap-and-hold on the marker to enable drag mode, then slide the marker around the map.

Note that at the present time, this functionality is a little odd. When you tap-and-hold the marker, with drag mode enabled, the marker initially jumps away from its original position. The user can reposition the marker to any desired location, and the marker will seem to "drop" where the user requests. Why the marker makes the sudden shift at the outset, using the default marker settings, is unclear.

Of course, your code may need to know about drag-and-drop events, such as to update your own data model to reflect the newly-chosen location. You can register an `OnMarkerDragListener` that will be notified of the start of the drag, where the marker slides during the drag, and where the marker is dropped at the end of the drag.

You can see all of this in the `MapsV2/Drag` sample application, which is a clone of `MapsV2/Popup` with drag-and-drop enabled.

To enable drag-and-drop, we just chain `draggable(true)` onto the series of calls on our `MarkerOptions` when creating the markers:

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet) {
  map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                   .title(getString(title))
                                   .snippet(getString(snippet))
                                   .draggable(true));
}
```

(from MapsV2/Drag/app/src/main/java/com/commonsware/android/mapsv2/drag/MainActivity.java)

We also register `MainActivity` as being the drag listener, up in `onMapReady()`:

```
  map.setOnMarkerDragListener(this);
```

(from MapsV2/Drag/app/src/main/java/com/commonsware/android/mapsv2/drag/MainActivity.java)

That requires `MainActivity` to implement `OnMarkerDragListener`, which in turn requires three methods to be defined: `onMarkerDragStart()`, `onMarkerDrag()`, and `onMarkerDragEnd()`:

```
@Override
public void onMarkerDragStart(Marker marker) {
  LatLng position=marker.getPosition();

  Log.d(getClass().getSimpleName(), String.format("Drag from %f:%f",
                                                  position.latitude,
                                                  position.longitude));
}

@Override
public void onMarkerDrag(Marker marker) {
  LatLng position=marker.getPosition();
```

```
    Log.d(getClass().getSimpleName(),
          String.format("Dragging to %f:%f", position.latitude,
                        position.longitude));
  }

  @Override
  public void onMarkerDragEnd(Marker marker) {
    LatLng position=marker.getPosition();

    Log.d(getClass().getSimpleName(), String.format("Dragged to %f:%f",
        position.latitude,
        position.longitude));
  }
```

(from MapsV2/Drag/app/src/main/java/com/commonsware/android/mapsv2/drag/MainActivity.java)

Here, we just dump the information about the new marker position in Logcat.

So, if you run this app and drag-and-drop a marker, you will see output in Logcat akin to:

```
12-19 13:10:36.442: D/MainActivity(22510): Drag from 40.770876:-73.982499
12-19 13:10:36.892: D/MainActivity(22510): Dragging to 40.770876:-73.981593
12-19 13:10:36.912: D/MainActivity(22510): Dragging to 40.770795:-73.981352
12-19 13:10:36.932: D/MainActivity(22510): Dragging to 40.770754:-73.981141
.
.
.
12-19 13:10:38.292: D/MainActivity(22510): Dragging to 40.769596:-73.983615
12-19 13:10:38.372: D/MainActivity(22510): Dragged to 40.769596:-73.983615
```

The actual list of events was much longer, as `onMarkerDrag()` is called a *lot*, so the . . . in the Logcat entries above reflect another 50 or so lines for a drag-and-drop that took a couple of seconds.

Also, up in `onCreate()`, we retain our `MapFragment` across configuration changes via `setRetainInstance(true)`:

```
      mapFrag.setRetainInstance(true);
```

(from MapsV2/Drag/app/src/main/java/com/commonsware/android/mapsv2/drag/MainActivity.java)

Retaining the fragment instance causes the fragment to keep our markers in their moved positions, rather than resetting them to their original positions.

The “Final” Limitations

In Maps V2, not only do you not create `Marker` objects directly yourself, but `Marker` is marked as `final` and cannot be extended. Hence, you cannot use a `Marker` directly to hold model data.

However, `Marker` does have `getId()`, an immutable identifier for the `Marker`. We can use that as a key for a `HashMap` that allows us to get at additional model data associated with the `Marker`.

You can see this approach in the `MapsV2/Models` sample application, which is a clone of `MapsV2/Popup` where we use the ID in just this fashion.

Our simplified model is merely the data we poured into our `Marker` objects in the original `MapsV2/Popup` project:

```
package com.commonsware.android.mapsv2.model;

import android.content.Context;

public class Model {
  String title;
  String snippet;
  double lat;
  double lon;

  Model(Context ctxt, double lat, double lon, int title,
        int snippet) {
    this.title=ctxt.getString(title);
    this.snippet=ctxt.getString(snippet);
    this.lat=lat;
    this.lon=lon;
  }

  String getTitle() {
    return(title);
  }

  String getSnippet() {
    return(snippet);
  }

  double getLatitude() {
    return(lat);
  }
```

```
  double getLongitude() {
    return(lon);
  }
}
```

(from MapsV2/Models/app/src/main/java/com/commonsware/android/mapsv2/model/Model.java)

Our activity holds onto a `HashMap` of these `Model` objects, with the map keyed by the `Marker` ID (a `String`):

```
  private HashMap<String, Model> models=new HashMap<String, Model>();
```

(from MapsV2/Models/app/src/main/java/com/commonsware/android/mapsv2/model/MainActivity.java)

Of course, a real application would have a much more elaborate setup than this.

We then arrange to populate our map with `Marker` objects created from our `Model` objects, moving the add-the-markers-to-the-map logic to an `addMarkers()` method:

```
  private void addMarkers(GoogleMap map) {
    Model model=
        new Model(this, 40.748963847316034, -73.96807193756104,
                  R.string.un, R.string.united_nations);

    models.put(addMarkerForModel(map, model).getId(), model);

    model=
        new Model(this, 40.76866299974387, -73.98268461227417,
                  R.string.lincoln_center,
                  R.string.lincoln_center_snippet);
    models.put(addMarkerForModel(map, model).getId(), model);

    model=
        new Model(this, 40.765136435316755, -73.97989511489868,
                  R.string.carnegie_hall, R.string.practice_x3);
    models.put(addMarkerForModel(map, model).getId(), model);

    model=
        new Model(this, 40.70686417491799, -74.01572942733765,
                  R.string.downtown_club, R.string.heisman_trophy);
    models.put(addMarkerForModel(map, model).getId(), model);
  }

  private Marker addMarkerForModel(GoogleMap map, Model model) {
    LatLng position=
        new LatLng(model.getLatitude(), model.getLongitude());
```

```
    return(map.addMarker(new MarkerOptions().position(position)
                                            .title(model.getTitle())
                                            .snippet(model.getSnippet())));

  }
```

(from MapsV2/Models/app/src/main/java/com/commonsware/android/mapsv2/model/MainActivity.java)

Notice that `addMarkerForModel()` returns the `Marker`, and we use `getId()` on that `Marker` as the key when adding a `Model` to the `HashMap`.

Our `PopupAdapter` gets the data for the info window from the `Model` (though, in truth, in this case, it could have gotten the data from the `Marker` itself, since we did not add more information to the info window):

```
package com.commonsware.android.mapsv2.model;

import android.view.LayoutInflater;
import android.view.View;
import android.widget.TextView;
import java.util.HashMap;
import com.google.android.gms.maps.GoogleMap.InfoWindowAdapter;
import com.google.android.gms.maps.model.Marker;

class PopupAdapter implements InfoWindowAdapter {
  LayoutInflater inflater=null;
  HashMap<String, Model> models=null;

  PopupAdapter(LayoutInflater inflater, HashMap<String, Model> models) {
    this.inflater=inflater;
    this.models=models;
  }

  @Override
  public View getInfoWindow(Marker marker) {
    return(null);
  }

  @Override
  public View getInfoContents(Marker marker) {
    View popup=inflater.inflate(R.layout.popup, null);

    TextView tv=(TextView)popup.findViewById(R.id.title);

    tv.setText(models.get(marker.getId()).getTitle());
    tv=(TextView)popup.findViewById(R.id.snippet);
```

```
    tv.setText(models.get(marker.getId()).getSnippet());

    return(popup);
  }
}
```

(from MapsV2/Models/app/src/main/java/com/commonsware/android/mapsv2/model/PopupAdapter.java)

Visually, this is indistinguishable from the original `MapsV2/Popups` project. Of course, a real app would have more complex models, perhaps containing more discrete information for a more complex info window.

A Bit More About IPC

IPC is not only a problem in terms of disappearing `Marker` objects.

If you run a Maps V2 app under Traceview, to see what methods get called and how much time everything takes, you will see that many, many operations with `GoogleMap` do little in your process, but instead make synchronous calls to a Play Services process to do the real work. You need to avoid manipulating your `GoogleMap` in time-sensitive portions of your code.

Finding the User

Many times, the user is looking at a map to figure out where they are. Perhaps they are lost. Perhaps their spouse or significant other thinks that they are lost. Perhaps they think that they were teleported somewhere (e.g., a North African desert) after turning a "frozen wheel" in an icy cavern beneath an island, and therefore are *really* lost. Stranger things have happened.

(well, OK, perhaps not)

Regardless, it is often useful to help point out to the user their current location. That is a matter of adding a suitable location permission (e.g., `ACCESS_FINE_LOCATION`) and calling `setMyLocationEnabled(true)` on your `GoogleMap`. This activates a layer that will highlight their location, with the user having an option of having the "camera" (i.e., their perspective on the map) reposition itself to their location and move as they move. This latter capability is activated by a small icon in the upper right of the map.

You can see this in operation in the MapsV2/MyLocation sample application, which is a clone of MapsV2/Popup with standard location tracking enabled.

All we do is call two additional methods on our GoogleMap in onCreate():

- setMyLocationEnabled(), indicating that we want the "my location" layer added and automatic tracking to be enabled, and
- setOnMyLocationChangeListener(), indicating that we *also* want to be notified about changes in the user position

```
    map.setMyLocationEnabled(true);
    map.setOnMyLocationChangeListener(this);
```

(from MapsV2/MyLocation/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

That latter method also requires our activity to implement the OnMyLocationChangeListener interface, which in turn requires us to implement the onMyLocationChange() method, which will be called when Maps V2 gets a new location fix:

```
@Override
public void onMyLocationChange(Location lastKnownLocation) {
  Log.d(getClass().getSimpleName(),
      String.format("%f:%f", lastKnownLocation.getLatitude(),
          lastKnownLocation.getLongitude()));
}
```

(from MapsV2/MyLocation/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

Here, we simply log the location to Logcat.

This is nice and easy, giving us our my-location overlay and arrow indicating the user's location and orientation:

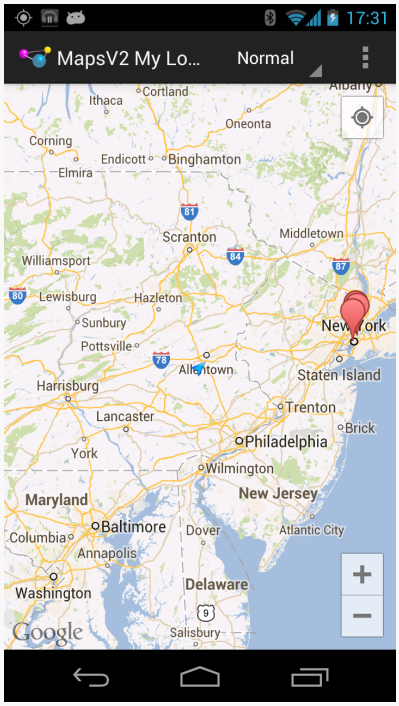


Figure 608: Maps V2, Showing the User's Location

However, there are three problems here. First, `setOnMyLocationChangeListener()` is now deprecated, as Google would prefer that you directly request the locations through the `LocationClient` available from Play Services.

Second, there does not appear to be a way to *force* camera tracking of the user's position — you are reliant upon the user tapping that icon. You also have no control over the nature of the location provider that is used.

However, there is a workaround for this, proposed in a Stack Overflow answer – provide your own location data and update the camera yourself, by means of `setLocationSource()`. `setLocationSource()` lets *you* push locations to the `GoogleMap`, making other adjustments (e.g., camera position) along the way.

To see how this works, take a peek at the `MapsV2/Location` sample application, which is a clone of `MapsV2/Popup` with custom location tracking enabled.

Along with adding `ACCESS_FINE_LOCATION` to the manifest, this sample project adds some lines to the `onMapReady()` implementation of `MainActivity` to configure the `GoogleMap`:

```
    locMgr=(LocationManager)getSystemService(LOCATION_SERVICE);
    crit.setAccuracy(Criteria.ACCURACY_FINE);
    locMgr.requestLocationUpdates(0L, 0.0f, crit, this, null);

    map.setLocationSource(this);
    map.setMyLocationEnabled(true);
    map.getUiSettings().setMyLocationButtonEnabled(false);
```

(from MapsV2/Location/app/src/main/java/com/commonsware/android/mapsv2/location/MainActivity.java)

The first three lines get access to a `LocationManager`, indicate that a `Criteria` object (initialized as a data member) should require fine accuracy, and request location updates. These come from the location tracking subsystem in Android.

The next two lines register our activity as being the source of location data and turns on location tracking in the `GoogleMap`, so the user's position will be marked on the map.

The last line disables the user's control over whether the camera position tracks their movement, since we want that to always be on in this case.

In `onResume()` and `onPause()` of `MainActivity`, we enable and disable getting location updates, as is typical of an activity needing location data. However, we also tell the `GoogleMap` that we are going to supply it with location data, rather than it having to obtain location data itself:

```
@Override
public void onResume() {
  super.onResume();

  if (locMgr!=null) {
    locMgr.requestLocationUpdates(0L, 0.0f, crit, this, null);
  }

  if (map!=null) {
    map.setLocationSource(this);
  }
}

@Override
public void onPause() {
  map.setLocationSource(null);
  locMgr.removeUpdates(this);
```

```
    super.onPause();
  }
```

(from MapsV2/Location/app/src/main/java/com/commonsware/android/mapsv2/location/MainActivity.java)

Note that we are blindly assuming that we will get location data. A production-grade app would put in better smarts to confirm that we will actually learn our location via this `Criteria` (e.g., the user does not have all location providers disabled).

Also note that because the map initialization is now happening on a background thread, `onResume()` might be called before `onMapReady()`, and so `onResume()` has to check to see if we already have a `LocationManager` and `GoogleMap` before proceeding.

The call to `setLocationSource()` — both in `onMapReady()` and `onResume()` – tells `GoogleMap` that our `MainActivity` itself is to be the source of location data. This requires `MainActivity` to implement the `LocationSource` interface, requiring us to implement `activate()` and `deactivate()` methods:

```
  @Override
  public void activate(OnLocationChangedListener listener) {
    this.mapLocationListener=listener;
  }

  @Override
  public void deactivate() {
    this.mapLocationListener=null;
  }
```

(from MapsV2/Location/app/src/main/java/com/commonsware/android/mapsv2/location/MainActivity.java)

`activate()` provides us with an `OnLocationChangedListener`, from `GoogleMap`, to which we need to pass location data as we get it. `deactivate()` indicates that we should no longer attempt to contact that listener. In addition to holding onto that listener (or removing our reference to it when deactivated), we also take this opportunity to request and remove location updates.

The `onLocationChanged()` method — where we get our location fixes from `LocationManager` via the `LocationListener` interface — must pass the location along to the `GoogleMap`-supplied `OnLocationChangedListener`, if we have such a listener available:

```
  @Override
  public void onLocationChanged(Location location) {
    if (mapLocationListener != null) {
```

```
    mapLocationListener.onLocationChanged(location);

    LatLng latlng=
        new LatLng(location.getLatitude(), location.getLongitude());
    CameraUpdate cu=CameraUpdateFactory.newLatLng(latlng);

    map.animateCamera(cu);
  }
}
```

(from MapsV2/Location/app/src/main/java/com/commonsware/android/mapsv2/location/MainActivity.java)

Here, we also create a `CameraUpdate` representing the new location and animate that update, to have the map slide over to the new location, centering the camera on the user's updated position.

The net effect of all of this is that the map continuously re-centers itself to show the user's position, which `GoogleMap` is highlighting on the map for us.

Dealing with Runtime Permissions

The previous section claimed three problems with the `MyLocation` sample, yet only explained two of them. That is because the third problem is shared by the `Location` sample as well: the apps are oblivious to Android 6.0's runtime permissions system.

Both samples have a `targetSdkVersion` of 22. They will install just fine on an Android 6.0 device, triggering the classic "accept all permissions at install time" dialog if installed by any means other than development tools. However, this also means that the apps will not know if the user goes into Settings and revokes the app's access to location data. Besides, *eventually* something will force your hand to have a `targetSdkVersion` of 23 or higher, and that will require you to adopt the new runtime permission system, whether you like it or not.

The `MapsV2/MyLocationMNC` sample application is nearly identical to the `MyLocation` sample, except that it has a `targetSdkVersion` of 23 and it makes limited use of the runtime permission system.

`onCreate()` of `MainActivity` now looks radically different:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
```

```
    if (savedInstanceState==null) {
      needsInit=true;
    }
    else {
      isInPermission=
        savedInstanceState.getBoolean(STATE_IN_PERMISSION, false);
    }

    onCreateForRealz(canGetLocation());
  }
```

(from MapsV2/MyLocationMNC/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

Most of the original business logic from `onCreate()` has been moved into `onCreateForRealz()`. That method takes a `boolean` parameter, indicating whether or not we have permission to access the user's location. Here, we get that from a `canGetLocation()` method that, in turn, uses `ContextCompat.checkSelfPermission()` to see if we hold `ACCESS_FINE_LOCATION`:

```
  private boolean canGetLocation() {
    return(ContextCompat.checkSelfPermission(this,
      Manifest.permission.ACCESS_FINE_LOCATION)==
      PackageManager.PERMISSION_GRANTED);
  }
```

(from MapsV2/MyLocationMNC/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

If we can work with locations, `onCreateForRealz()` will do what `onCreate()` used to do: call `readyToGo()` and, if we are ready to go, bring up the map:

```
  private void onCreateForRealz(boolean canGetLocation) {
    if (canGetLocation) {
      if (readyToGo()) {
        setContentView(R.layout.activity_main);

        MapFragment mapFrag=
          (MapFragment)getFragmentManager().findFragmentById(
            R.id.map);

        mapFrag.getMapAsync(this);
      }
    }
    else if (!isInPermission) {
      isInPermission=true;

      ActivityCompat.requestPermissions(this,
        new String[] {Manifest.permission.ACCESS_FINE_LOCATION},
```

```
        REQUEST_PERMS);
    }
  }
```

(from MapsV2/MyLocationMNC/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

If we do not have access to the user's location, this particular sample app is not that interesting, so we will ask the user for permission, via a call to `ActivityCompat.requestPermissions()`. This will eventually trigger a call to `onRequestPermissionsResult()`:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  isInPermission=false;

  if (requestCode==REQUEST_PERMS) {
    if (canGetLocation()) {
      onCreateForRealz(true);
    }
    else {
      finish(); // denied permission, so we're done
    }
  }
}
```

(from MapsV2/MyLocationMNC/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

Here, if we can now get the location, we go ahead and run through `onCreateForRealz()` again, to initialize the map. If, however, the user denied us the right to access the location, we `finish()` and exit the activity outright. One could argue that a better approach would be to show the map and simply not call `setMyLocationEnabled(true)` or `setOnMyLocationChangeListener()`. That would be another approach to dealing with the missing permission, and it is probably the better option if your primary goal was to just show a map.

Throughout this code, you have seen references to an `isInPermission` field. This tracks whether or not we are in the middle of requesting a permission:

- It is initialized to `false` in the activity
- It is set to `true` just before calling `requestPermissions()`
- It is set back to `false` in `onRequestPermissionsResult()`
- It is saved across configuration changes via `onSaveInstanceState()` and is retrieved from that state in `onCreate()`:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putBoolean(STATE_IN_PERMISSION, isInPermission);
}
```

(from MapsV2/MyLocationMNC/app/src/main/java/com/commonsware/android/mapsv2/mylocation/MainActivity.java)

(where `STATE_IN_PERMISSION` is a `static final String` to use as a key for the `Bundle` value)

This allows us to check whether or not we are in the middle of requesting permissions already in `onCreateForRealz()` and avoid popping up the permission-request dialog twice if the user rotates the screen while the first dialog is up, then denies the permission.

Also note that we are not taking any steps here to leverage `ActivityCompat.showShowRequestPermissionRationale()`, in case the user denied the permission on some previous run of our app, but then ran the app again. You could do something for that here, such as pop up a dialog and call `requestPermissions()` afterwards.

Drawing Lines and Areas

If you wanted to draw on a map in the Maps V1 framework, you created an `Overlay` and drew upon it. This forced you to handle low-level drawing work yourself, as you were handed a `Canvas` object and had to handle all the lines, fills, and so forth yourself.

Maps V2 offers a different approach. Free-form drawing is still conceivable, though it appears to have to be handled in the form of tile overlays instead of map overlays. However, for the simpler cases of drawing lines and areas, Maps V2 has built-in polyline, polygon, and circle support. You tell the `GoogleMap` what needs to be drawn, and it handles drawing it, both initially and as the map is zoomed or panned. A polyline is a line connecting a series of points; a polygon is a region defined by a series of corners. A circle, from the standpoint of Maps V2, is defined by a center coordinate and a radius.

We can see polylines and polygons on a `GoogleMap` in the `MapsV2/Poly` sample application, which is a clone of `MapsV2/Popup` with two additions:

- A polyline connecting the locations of our four markers
- A polygon enclosing the area of Manhattan known as the Garment District (bounded by 34th Street, 42nd Street, Fifth Avenue, and Ninth Avenue)

To draw those, we simply add a few lines to `onMapReady()` of `MainActivity`:

```
    PolylineOptions line=
        new PolylineOptions().add(new LatLng(40.70686417491799,
                                             -74.01572942733765),
                                  new LatLng(40.76866299974387,
                                             -73.98268461227417),
                                  new LatLng(40.765136435316755,
                                             -73.97989511489868),
                                  new LatLng(40.748963847316034,
                                             -73.96807193756104))
                              .width(5).color(Color.RED);

    map.addPolyline(line);

    PolygonOptions area=
        new PolygonOptions().add(new LatLng(40.748429, -73.984573),
                                 new LatLng(40.753393, -73.996311),
                                 new LatLng(40.758393, -73.992705),
                                 new LatLng(40.753484, -73.980882))
                            .strokeColor(Color.BLUE);

    map.addPolygon(area);
```

(from MapsV2/Poly/app/src/main/java/com/commonsware/android/mapsv2/poly/MainActivity.java)

The API for adding polylines and polygons is reminiscent of the API for adding markers: define an `...Options` object with the characteristics of the item to be drawn, then call an `add...()` method on `GoogleMap` to add the item.

So, to add a polyline, we create a `PolylineOptions` object. Using its fluent interface, we `add()` a series of `LatLng` objects, representing the points to be connected by the line. We also specify the line width in *pixels* via `width()` and the color of the line via `color()`. If we had several lines that might overlap, we could specify the `zIndex()`, where higher indexes indicate lines to be drawn over the top of lines with lower indexes. We add the polyline to the map by passing our `PolylineOptions` to `addPolyline()` on `GoogleMap`.

This gives us a line connecting the four markers, with `GoogleMap` handling the details of where the line should be drawn on the screen given the current map center and zoom levels:

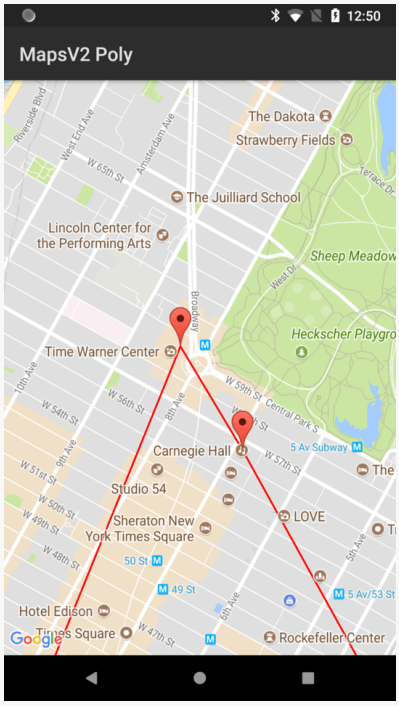


Figure 609: Maps V2 with Polyline

Note that the polyline is drawn using a flat Mercator projection by default. For most maps, that is perfectly fine. If your map will be showing countries and continents, rather than city blocks, you might want to call `geodesic(true)` on the `PolylineOptions`, to have the line drawn on a geodesic curve, reflecting the spherical nature of the Earth (dissenting opinions on that notwithstanding).

Similarly, we create a `PolygonOptions` object, configure it, and pass it to `addPolygon` for our Garment District box. The `add()` method on `PolygonOptions` will take the corners of our polygon, automatically enclosing that region. We also specify the `strokeColor()`. We could have specified a `fillColor()` (default is transparent), `strokeWidth()` (default is 10 pixels), `zIndex()`, and `geodesic()`.

If we run the app and pan the map down to the south a bit, we see our polygon:

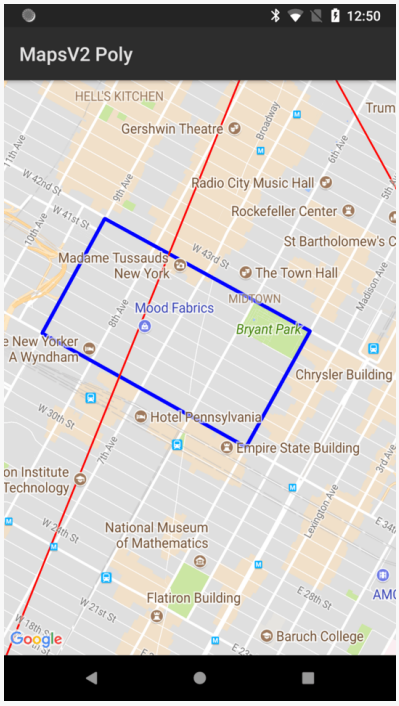


Figure 610: Maps V2 with Polyline and Polygon

As with the polyline, Android automatically handles drawing what is needed based on map center and zoom levels.

Note that, as with markers, we need to re-add the polylines and polygons after a configuration change, as the `GoogleMap` does not retain that information.

Gestures and Controls

By default, standard gestures and controls are enabled on your map:

- The user can change zoom level either by + and - buttons or via “pinch-to-zoom” gestures
- The user can change the center of the map via simple swipe gestures
- The user can change the camera tilt via two-finger vertical swipes, so instead of a traditional top-down perspective, the user can see things on an angle
- The user can change the orientation of the map via a two-finger rotating swipe, to change the typical “north is to the top of the map” to some other orientation

You can obtain a `UiSettings` object from your `GoogleMap` via `getUiSettings()` to disable these features, if desired:

- `setRotateGesturesEnabled()`
- `setScrollGesturesEnabled()` (for panning the map)
- `setTiltGesturesEnabled()`
- `setZoomControlsEnabled()` (for the + and - buttons)
- `setZoomGesturesEnabled()` (for pinch-to-zoom)

There is also `setAllGesturesEnabled()` to toggle on or off all gesture-based map control. This is roughly analogous to the `android:clickable` attribute on the Maps V1 edition of `MapView`.

There is also `setCompassEnabled()`, to indicate if a compass should be shown if the user changes the map orientation via a rotate gesture.

Tracking Camera Changes

If you have gestures enabled, the user can change the perspective of the map, referred to as changing the camera position. You may need to know about these changes, to perform various operations in your app based upon what is presently visible on the screen.

Originally, to find out when the camera position changes, you could call `setOnCameraChangeListener()` on the `GoogleMap`, supplying an implementation of `OnCameraChangeListener`, which would be called with `onCameraChange()` as the user pans, zooms, or tilts the map. This approach was deprecated and replaced with a series of listeners:

- `OnCameraMoveStartedListener`, invoked when the user starts moving the map
- `OnCameraMoveListener`, invoked when the user continues moving the map after having originally started moving it, all in one gesture
- `OnCameraIdleListener`, invoked when the user stops moving the map (e.g., lifts up their finger or stylus)
- `OnCameraMoveCanceledListener`, invoked if you do something programmatically to interrupt the camera movement

To see how this works, we can take a quick peek at the MapsV2/Camera sample application, which is a clone of MapsV2/Popup with camera position tracking enabled.

Late in onMapReady() of MainActivity, we call a series of setter methods on the GoogleMap to associate MainActivity itself as the listener for these events:

```
    map.setOnCameraMoveStartedListener(this);
    map.setOnCameraMoveListener(this);
    map.setOnCameraMoveCanceledListener(this);
    map.setOnCameraIdleListener(this);
```

(from MapsV2/Camera/app/src/main/java/com/commonsware/android/mapsv2/camera/MainActivity.java)

This requires MainActivity to implement all four of those listener interfaces:

```
public class MainActivity extends AbstractMapActivity implements
    OnMapReadyCallback, OnInfoWindowClickListener,
  OnCameraMoveStartedListener,
  OnCameraMoveListener,
  OnCameraMoveCanceledListener,
  OnCameraIdleListener {
```

(from MapsV2/Camera/app/src/main/java/com/commonsware/android/mapsv2/camera/MainActivity.java)

And, this requires MainActivity to implement the callback method for each of those listeners:

Listener Interface	Event Method
OnCameraMoveStartedListener	onCameraMoveStarted(int i)
OnCameraMoveListener	onCameraMove()
OnCameraIdleListener	onCameraIdle()
OnCameraMoveCanceledListener	onCameraMoveCanceled()

```
  @Override
  public void onCameraIdle() {
    CameraPosition position=map.getCameraPosition();

    Log.d("onCameraIdle",
      String.format("lat: %f, lon: %f, zoom: %f, tilt: %f",
        position.target.latitude,
        position.target.longitude, position.zoom,
        position.tilt));
  }
```

```
@Override
public void onCameraMoveCanceled() {
  CameraPosition position=map.getCameraPosition();

  Log.d("onCameraMoveCanceled",
    String.format("lat: %f, lon: %f, zoom: %f, tilt: %f",
      position.target.latitude,
      position.target.longitude, position.zoom,
      position.tilt));
}

@Override
public void onCameraMove() {
  CameraPosition position=map.getCameraPosition();

  Log.d("onCameraMove",
    String.format("lat: %f, lon: %f, zoom: %f, tilt: %f",
      position.target.latitude,
      position.target.longitude, position.zoom,
      position.tilt));
}

@Override
public void onCameraMoveStarted(int i) {
  CameraPosition position=map.getCameraPosition();

  Log.d("onCameraMoveStarted",
    String.format("lat: %f, lon: %f, zoom: %f, tilt: %f",
      position.target.latitude,
      position.target.longitude, position.zoom,
      position.tilt));
}
```

(from MapsV2/Camera/app/src/main/java/com/commonsware/android/mapsv2/camera/MainActivity.java)

Here, we just log a message to Logcat on each camera position change, logging:

- the latitude and longitude of the map center, obtained from the `target` `LatLng` data member of the `CameraPosition` object supplied to `onCameraChange()`,
- the zoom level of the map, from the `zoom` data member of `CameraPosition`, and
- the tilt of the map, in degrees, from the `tilt` data member of `CameraPosition`

As a result, if you run this app and play around with the various gestures, you get a series of Logcat messages with the results:

```
10-08 12:13:54.449 28718-28718/com.commonsware.android.mapsv2.camera D/
onCameraMoveStarted: lat: 40.771664, lon: -73.986067, zoom: 15.000000, tilt: 0.000000
10-08 12:13:54.453 28718-28718/com.commonsware.android.mapsv2.camera D/onCameraMove:
lat: 40.771773, lon: -73.985717, zoom: 15.000000, tilt: 0.000000
10-08 12:13:54.483 28718-28718/com.commonsware.android.mapsv2.camera D/onCameraMove:
lat: 40.771805, lon: -73.985701, zoom: 15.000000, tilt: 0.000000
10-08 12:13:54.500 28718-28718/com.commonsware.android.mapsv2.camera D/onCameraMove:
lat: 40.771843, lon: -73.985669, zoom: 15.000000, tilt: 0.000000
.
.
.
10-08 12:13:57.001 28718-28718/com.commonsware.android.mapsv2.camera D/onCameraIdle:
lat: 40.774540, lon: -73.985632, zoom: 15.000000, tilt: 0.000000
```

Note that `onCameraMoveStarted()` will be invoked for three reasons:

1. The user started panning, tilting, or rotating the map, or used a pinch-to-zoom gesture
2. The user did something else that triggered a camera change, such as tapping the "my location" button to move the camera to their current location
3. You did something programmatically to change the camera position

The parameter passed into `onCameraMoveStarted()` will contain a reason code (e.g., `REASON_GESTURE`) to help you distinguish these cases, if that level of detail is needed by your app.

Maps in Fragments and Pagers

One key limitation of Maps V1 was that you could only have one `MapView` instance per *process*. Presumably, the proprietary code at the heart of the Maps SDK add-on used static data members for some state management, ones that would get messed up if there were two or more `MapView` widgets in active use.

Fortunately, Maps V2 gets rid of this restriction. You are welcome to have multiple `MapFragment` objects if that makes sense. Maps are relatively memory-intensive, so you should not be planning on having dozens or hundreds of them in use at a time, but you can have more than one.

To showcase this, the MapsV2/Pager sample application hosts 10 MapFragment instances as pages in a ViewPager. The bulk of the application is a clone of one of the ViewPager samples from the chapter on ViewPager.

Having maps in a ViewPager presents a bit of a problem, in terms of interpreting horizontal swipe events. Normally, ViewPager handles those itself. However, that would mean that the user cannot pan the map horizontally, which makes using the map somewhat challenging. In this sample, we will augment the ViewPager with logic to allow horizontal swiping on the maps and on the tab strip.

Our activity inflates a layout that contains our ViewPager along with a PagerTabStrip:

```
<?xml version="1.0" encoding="utf-8"?>
<com.commonsware.android.mapsv2.pager.MapAwarePager xmlns:android="http://schemas.android.com/apk/res/
android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>

</com.commonsware.android.mapsv2.pager.MapAwarePager>
```

(from MapsV2/Pager/app/src/main/res/layout/activity_main.xml)

However, you will note that this is not ViewPager, but rather MapAwarePager, a custom subclass of ViewPager that we will examine shortly.

MainActivity then populates the MapAwarePager with an instance of a MapPageAdapter:

```
package com.commonsware.android.mapsv2.pager;

import android.os.Bundle;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;

public class MainActivity extends AbstractMapActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
      setContentView(R.layout.activity_main);
```

```
      ViewPager pager=(ViewPager)findViewById(R.id.pager);

      pager.setAdapter(buildAdapter());
    }
  }

  private PagerAdapter buildAdapter() {
    return(new MapPageAdapter(this, getFragmentManager()));
  }
}
```

(from MapsV2/Pager/app/src/main/java/com/commonsware/android/mapsv2/pager/MainActivity.java)

MapPageAdapter is a FragmentStatePagerAdapter, not a FragmentPagerAdapter. This means that as the user swipes through our ViewPager, the adapter has the right to discard old fragments when it creates new ones. This helps reduce the overall memory footprint of our activity.

```
package com.commonsware.android.mapsv2.pager;

import android.content.Context;
import android.app.Fragment;
import android.app.FragmentManager;
import android.support.v13.app.FragmentStatePagerAdapter;

public class MapPageAdapter extends FragmentStatePagerAdapter {
  Context ctxt=null;

  public MapPageAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(10);
  }

  @Override
  public Fragment getItem(int position) {
    return(new PageMapFragment());
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(R.string.map_page_title) + String.valueOf(position + 1));
  }
}
```

(from MapsV2/Pager/app/src/main/java/com/commonsware/android/mapsv2/pager/MapPageAdapter.java)

MapPageAdapter declares that there should be ten pages (in getCount()) and returns an instance of PageMapFragment for each page. PageMapFragment is a subclass of MapFragment, and so is responsible for displaying our map:

```
package com.commonsware.android.mapsv2.pager;

import android.os.Bundle;
import com.google.android.gms.maps.CameraUpdate;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.MapFragment;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class PageMapFragment extends MapFragment implements
    OnMapReadyCallback {
  private boolean needsInit=false;

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    if (savedInstanceState == null) {
      needsInit=true;
    }

    getMapAsync(this);
  }

  @Override
  public void onMapReady(final GoogleMap map) {
    if (needsInit) {
      CameraUpdate center=
          CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044,
                                                   -73.98180484771729));
      CameraUpdate zoom=CameraUpdateFactory.zoomTo(15);

      map.moveCamera(center);
      map.animateCamera(zoom);
    }

    addMarker(map, 40.748963847316034, -73.96807193756104, R.string.un,
              R.string.united_nations);
    addMarker(map, 40.76866299974387, -73.98268461227417,
              R.string.lincoln_center, R.string.lincoln_center_snippet);
    addMarker(map, 40.765136435316755, -73.97989511489868,
```

```
              R.string.carnegie_hall, R.string.practice_x3);
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy);
  }

  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet) {
    map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                     .title(getString(title))
                                     .snippet(getString(snippet)));
  }
}
```

(from MapsV2/Pager/app/src/main/java/com/commonsware/android/mapsv2/pager/PageMapFragment.java)

If we simply wanted to display an unconfigured map, we could just have `MapPageAdapter` create and return instances of `MapFragment` directly. If we want to configure our map, though, we need to get control when the `GoogleMap` object is ready for use. One way to do that is to extend `MapFragment` and override `onActivityCreated()` and call `getMapAsync()` there to begin the whole get-the-`GoogleMap`-loaded process. In `onMapReady()`, we can then go ahead and configure the map much as we have done in previous examples, just from within the fragment itself rather than from the hosting activity.

`MapAwarePager` overrides one key method of `ViewPager`: `canScroll()`:

```
package com.commonsware.android.mapsv2.pager;

import android.content.Context;
import android.support.v4.view.PagerTabStrip;
import android.support.v4.view.ViewPager;
import android.util.AttributeSet;
import android.view.SurfaceView;
import android.view.View;

public class MapAwarePager extends ViewPager {
  public MapAwarePager(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override
  protected boolean canScroll(View v, boolean checkV, int dx, int x,
                              int y) {
    if (v instanceof SurfaceView || v instanceof PagerTabStrip) {
      return(true);
    }
```

```
    return(super.canScroll(v, checkV, dx, x, y));
  }
}
```

(from MapsV2/Pager/app/src/main/java/com/commonsware/android/mapsv2/pager/MapAwarePager.java)

`canScroll()` should return `true` if the `View` (and specifically the supplied X and Y coordinates within that `View`) can be scrolled horizontally, `false` otherwise. In our case, we want to say that the map and the tab strip are each scrollable horizontally. As it turns out, the passed-in `View` for our `MapFragment` will be the map if it is a subclass of `SurfaceView` (determined by trial and error on the author's part, with hopes for a more authoritative solution in a future edition of the Maps V2 API). So, if the passed-in `View` is either a `SurfaceView` or a `PagerTabStrip`, we return `true`, otherwise we default to normal logic.

The result is a series of independent maps, one per page:

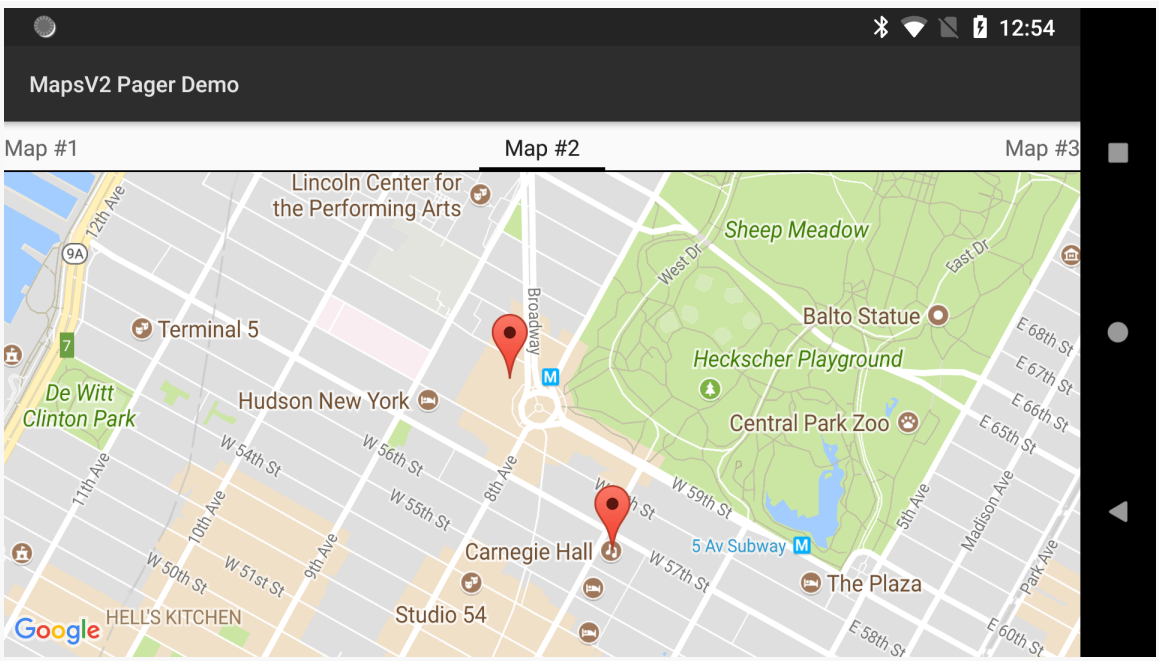


Figure 611: Multiple Maps V2 Maps in a ViewPager

Each map is independent: if the user pans or zooms one map, that has no impact on any of the other pages. Panning the maps horizontally works; to move between pages, use the tab strip.

Animating Marker Movement

Markers, by default, are static, unless you make them be draggable, and then only the user can drag them.

However, you are welcome to update the position of a `Marker` at any point, by calling `setPosition()` and supplying a new `LatLng`. The `Marker` then will jump to that position.

But what if you want to animate the movement of a `Marker` from its current position to a new one? Maps V2 does not offer anything "out of the box" for implementing this, but Google demonstrated approaches for this in a "DevBytes" video and related bit of code in a GitHub Gist. This section will cover the technique appropriate for API Level 14+, including a full working sample (the Gist shows code but not its usage).

Problem #1: Animating a LatLng

The position of a `Marker` is a `LatLng`, as we have seen previously. `LatLng` is not a simple number, and so the animator framework needs our assistance to animate them. Specifically, we need a `TypeEvaluator` for `LatLng`, with our `evaluate()` method taking the initial and end positions and computing another `LatLng` representing the `fraction` position between those other positions. This concept was introduced back in the chapter on the animator framework.

A simple approach to computing the fractional `LatLng` would be to apply the fraction to the latitude and the longitude as Java double values:

```
LatLng interpolate(float fraction, LatLng initial, LatLng end) {
  double lat = (end.latitude - initial.latitude) * fraction + initial.latitude;
  double lng = (end.longitude - initial.longitude) * fraction + initial.longitude;

  return(new LatLng(lat, lng));
}
```

That would work reasonably well for fairly close points, such as animating a marker within a city. However, animating markers across longer distances means that we have to take into account some geographic realities that a simple calculation will miss.

Problem #2: The Earth Is Not Flat (Really!)

One bit of reality is that the Earth is round. The above calculation assumes that the Earth is flat. Calculating "great circle" positions requires a fair bit of spherical trigonometry, known to cause loss of hair in software developers.

Hence, ideally, we will use somebody's existing debugged algorithm for that.

Problem #3: 180 Equals –180, At Least For Longitude

The other problem is that longitudes wrap around, as 180 degrees longitude is equivalent to –180 degrees longitude, and longitudinal values are considered to be between 180 and –180. In cases where we would not cross 180 degrees longitude, this is not an issue. However, a simple calculation might miss this and wind up having our animation "take the long way" (e.g., animating from –175 degrees longitude to 175 degrees longitude by going 350 degrees around the Earth, rather than just 10 degrees and crossing the International Date Line).

Introducing Some Googly Assistance

Google themselves have released a utility library for Maps V2. It offers polyline and polygon decoding, primarily for interoperability with other location-related Google services like the Google Directions API. The `SphericalUtil` class handles all of the nasty math for computing distances along the surface of the Earth and related calculations. It also offers `BubbleIconFactory`, which makes it easy to create marker icons that look a bit like info windows (complete with border and caret) wrapping around a bit of text or an icon.

In our case, we can use `SphericalUtil` to handle Problem #2 and Problem #3, interpolating the location between two `LatLng` values, taking the curvature of the Earth and longitude idiosyncrasies into account.

Seeing This in Action

The `MapsV2/Animator` sample project is a modified version of the `MapsV2/Markers` project, adding in the notion of animating a marker from its original position (Lincoln Center) to a new position (Penn Station) within Manhattan.

Since we want to use the Google map utility library, we need to add it as a dependency. Android Studio users can simply add `implementation`

`'com.google.maps.android:android-maps-utils:0.3.4'` (or a higher version) to the `dependencies` closure.

We need to know where our starting and ending position for the animation will be, in terms of `LatLng` objects. Since those have no dependencies upon a `Context` or anything, we can simply declare them as `static final` values:

```
private static final LatLng PENN_STATION=new LatLng(40.749972,
                                                    -73.992319);
private static final LatLng LINCOLN_CENTER=
    new LatLng(40.76866299974387, -73.98268461227417);
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

We will also need the actual `Marker` object created when we add our starting position (`LINCOLN_CENTER`) to the map. So far, we have ignored the `Marker` returned by `addMarker()` on `GoogleMap`, but now we need that. So, our own `addMarker()` method now returns this value:

```
private Marker addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet) {
  return(map.addMarker(new MarkerOptions().position(new LatLng(lat,
                                                                lon))
                                          .title(getString(title))
                                          .snippet(getString(snippet))));
}
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

We also now have a `markerToAnimate` data member of the activity, for our `Marker`, which we populate from our modified `addMarker()` method:

```
  addMarker(map, 40.748963847316034, -73.96807193756104,
            R.string.un, R.string.united_nations);
  markerToAnimate=
      addMarker(map, LINCOLN_CENTER.latitude,
                LINCOLN_CENTER.longitude, R.string.lincoln_center,
                R.string.lincoln_center_snippet);
  addMarker(map, 40.765136435316755, -73.97989511489868,
            R.string.carnegie_hall, R.string.practice_x3);
  addMarker(map, 40.70686417491799, -74.01572942733765,
            R.string.downtown_club, R.string.heisman_trophy);
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

To make the sample work repeatedly, it would be nice to support bi-directional animation, starting with animating from Lincoln Center to Penn Station, then reversing the animation to go back to Lincoln Center. That means that we need to know, for any particular animation, where the end position should be. So, we track a `LatLng` for the next end position, surprisingly named `nextAnimationEnd`, initializing it to be `PENN_STATION` (since we are starting at the outset at `LINCOLN_CENTER`):

```
  private LatLng nextAnimationEnd=PENN_STATION;
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

Next, we need to give the user a means of actually requesting the animation to run. To do that, we define a new menu XML resource for an `animate` menu item (using the `directions` icon for lack of a better handy icon):

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/animate"
    android:icon="@android:drawable/ic_menu_directions"
    android:showAsAction="ifRoom"
    android:title="@string/animate"/>

</menu>
```

(from MapsV2/Animator/app/src/main/res/menu/animate.xml)

We then load that menu resource in an overridden `onCreateOptionsMenu()` and direct the click event to an `animateMarker()` method in `onOptionsItemSelected()`:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.animate, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.animate) {
      animateMarker();

      return(true);
    }
```

```
    return(super.onOptionsItemSelected(item));
  }
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

In `animateMarker()`, we need to do two things:

1. Actually run the animation
2. Ensure that the camera position is such that the animation will actually be visible, as it is pointless to animate a marker between two points if the currently-viewed portion of the map does not show those points

To handle the camera position, we need to use `moveCamera()` with a `CameraUpdate` from `CameraUpdateFactory`, as we used to set the initial camera position and zoom level. To handle the case where we want one or more points to be visible, we can use the `newLatLngBounds()` method on `CameraUpdateFactory`. This takes a `LatLngBounds` describing the area that needs to be visible, plus a padding amount in pixels for where that area should be inset within the map.

Of course, this implies that we have a `LatLngBounds`.

Since `LatLngBounds` also does not depend upon a `Context` or much of anything, we can define one of those as a `static final` data member, using a `LatLngBounds.Builder` instance:

```
  private static final LatLngBounds bounds=
      new LatLngBounds.Builder().include(LINCOLN_CENTER)
                                .include(PENN_STATION).build();
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

A `LatLngBounds.Builder` takes one or more `LatLng` objects — passed in via `include()` – then constructs a `LatLngBounds` that encompasses all of those points via `build()`.

Our `animateMarker()` method then starts off by using `moveCamera()` to reset the camera to show that defined region:

```
  private void animateMarker() {
    map.moveCamera(CameraUpdateFactory.newLatLngBounds(bounds, 48));

    Property<Marker, LatLng> property=
        Property.of(Marker.class, LatLng.class, "position");
    ObjectAnimator animator=
```

```
      ObjectAnimator.ofObject(markerToAnimate, property,
                              new LatLngEvaluator(), nextAnimationEnd);
  animator.setDuration(2000);
  animator.start();

  if (nextAnimationEnd == LINCOLN_CENTER) {
    nextAnimationEnd=PENN_STATION;
  }
  else {
    nextAnimationEnd=LINCOLN_CENTER;
  }
}
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

Then, we need to set up the animation. To do this, we will use the object animator framework, specifically an `ObjectAnimator`. We know the `Marker` that we want to animate (`markerToAnimate`) and we know where we want to animate it to (`nextAnimationEnd`). What we need is to indicate the property to animate on this object, plus provide help to actually animate a `LatLng`.

To specify the property, we could just pass in the name of the property (`"position"`). However, in `animateMarker()`, we set up a `Property` object via the static `of()` factory method. This makes our use of `ofObject()` more type-safe, as `Property` will help enforce that we are animating a `Marker` using `LatLng` values.

To animate `LatLng` values, we need a `TypeEvaluator` for `LatLng`, here defined as a static inner class named `LatLngEvaluator`:

```
private static class LatLngEvaluator implements TypeEvaluator<LatLng> {
  @Override
  public LatLng evaluate(float fraction, LatLng startValue,
                         LatLng endValue) {
    return(SphericalUtil.interpolate(startValue, endValue, fraction));
  }
}
```

(from MapsV2/Animator/app/src/main/java/com/commonsware/android/mapsv2/animator/MainActivity.java)

Our `evaluate()` method turns around and calls the static `interpolate()` method on `SphericalUtil`, supplied by Google's map utility library. `interpolate()` handles all the nasty spherical trigonometry and stuff, so we do not have to.

We then set the duration of the animation to be two seconds, and start the animation.

Finally, to reverse the animation for the next request, `animateMarker()` resets the value of `nextAnimationEnd` to be `PENN_STATION` or `LINCOLN_CENTER`, wherever we will animate to next.

Honoring Traffic Rules, Like "Drive Only On Streets"

You will notice that our animation ignores other aspects of reality, such as buildings that might be in the way. Sometimes, that is appropriate, such as animating the movement of:

- a bird
- a plane
- a costumed superhero with independent flight capability

Sometimes, though, we need to take into account those obstacles, such as animating the movement of:

- a pedestrian
- a car
- a costumed superhero "flying" by means of swinging between buildings using dynamically-generated cables of either natural or synthetic origin

However, to do this implies that we know where the obstacles are. Or, more accurately, we would need to animate the marker along known good waypoints, such as streets.

The animation would not be especially difficult, as `ofObject()` can take a series of waypoints. However, we would need to find those waypoints, and there is nothing in Maps V2 itself that supplies this data.

Maps, of the Indoor Variety

The good news is that Maps V2 supports Google's indoor maps, for those venues for which Google has indoor map data.

The bad news is that for some reason, only one map at a time supports indoor maps. The default will be that the first map you create will support indoor maps, and others will not.

To see if a given map offers indoor map capability, you can call `isIndoorEnabled()` on `GoogleMap`. To toggle this capability, call `setIndoorEnabled()`.

Taking a Snapshot of a Map

Once a map is drawn, you can take a snapshot of it, converting the viewed map into a `Bitmap` object. This is designed to take an image of the map and use it in places where a `MapFragment`, or even a `MapView`, cannot go, such as:

- Things tied to a `RemoteViews`, such as a custom `Notification`
- Thumbnails of maps, for an app that allows users to manipulate several maps at once

The `GoogleMap` object has two flavors of a `snapshot()` method. Both take a `SnapshotReadyCallback` object. You will need to supply an instance of something implementing the `SnapshotReadyCallback` interface, overriding `onSnapshotReady()`, where you will receive your `Bitmap`.

One flavor of `snapshot()` takes just the `SnapshotReadyCallback`; the other also takes a `Bitmap` of the proper dimensions, such as a previous snapshot `Bitmap` that you want to recycle. Using the latter `snapshot()` is recommended where possible, so you do not need to allocate new `Bitmap` objects on each `snapshot()` call.

Note that `snapshot()` will only work once the map is actually rendered. So, for example, calling `snapshot()` from `onCreate()` of your activity will fail, because the map has not been rendered yet. `snapshot()` is designed to be called based upon user input, either to manually capture a snapshot or based on navigation (e.g., tapping on a `ListView` item triggers saving a snapshot of the current map as a thumbnail before changing the map contents).

Also, the documentation for `snapshot()` contains the following:

> **Note**: Images of the map must not be transmitted to your servers, or otherwise used outside of the application. If you need to send a map to another application or user, send data that allows them to reconstruct the map for the new user instead of a snapshot.

As this statement may be tied to the terms and conditions of your use of Maps V2, you should talk with qualified legal counsel before:

- Saving a snapshot to external storage
- Sharing a snapshot via `ACTION_SEND`
- Sending a snapshot to your server

or similar operations.

MapFragment vs. MapView

So far, all the examples shown in this chapter use `MapFragment`. In most cases, this is the right thing to use.

However, there may be places where you really want to use a `View`, rather than a `Fragment`, for your maps.

The good news is that Maps V2 does have a `MapView`. `MapFragment` usually handles creating and managing the `MapView` for you, but you can, if you wish, avoid `MapFragment` and manage the `MapView` yourself.

The biggest limitation is that you need to forward the lifecycle methods from your activity or fragment on to the `MapView`, calling `onCreate()`, `onResume()`, `onPause()`, `onDestroy()`, and `onSaveInstanceState()` on the `MapView`. Normally, `MapFragment` would do that for you, saving you the trouble.

Also note that while `MapView` is a `ViewGroup`, you are not allowed to add child widgets to it.

About That AbstractMapActivity Class…

Early on, we hand-waved our way past the `AbstractMapActivity` that all of our `MainActivity` classes inherit from, and we skirted past the `readyToGo()` method that we were calling. Also, you may have noticed that our app has an action bar overflow item, that we do not seem to be creating in `MainActivity`.

Now, it is time to dive into what is going on in our `AbstractMapActivity` implementations.

The `readyToGo()` method in `AbstractMapActivity` is designed to help us determine if Maps V2 is "ready to go" and, if not, to help the user perhaps fix their device such that Maps V2 will work in the future:

```
protected boolean readyToGo() {
  GoogleApiAvailability checker=
    GoogleApiAvailability.getInstance();

  int status=checker.isGooglePlayServicesAvailable(this);

  if (status == ConnectionResult.SUCCESS) {
    if (getVersionFromPackageManager(this)>=2) {
      return(true);
    }
    else {
      Toast.makeText(this, R.string.no_maps, Toast.LENGTH_LONG).show();
      finish();
    }
  }
  else if (checker.isUserResolvableError(status)) {
    ErrorDialogFragment.newInstance(status)
                       .show(getFragmentManager(),
                             TAG_ERROR_DIALOG_FRAGMENT);
  }
  else {
    Toast.makeText(this, R.string.no_maps, Toast.LENGTH_LONG).show();
    finish();
  }

  return(false);
}
```

(from MapsV2/Basic/app/src/main/java/com/commonsware/android/mapsv2/basic/AbstractMapActivity.java)

Determining the availability of Maps V2 — or anything in the Play Services SDK — is handled through an instance of `GoogleApiAvailability`. You get a singleton instance of this class via its `static` `getInstance()` method.

First, we call `isGooglePlayServicesAvailable()` method on the `GoogleApiAvailability` singleton. This will return an integer indicating whether Maps V2 is available for our use or not.

If the return value is `ConnectionResult.SUCCESS` — meaning Maps V2 is indeed available to us – we check to see if OpenGL ES is version 2.0 or higher, as we did not require that in the manifest. There are a few ways in Android to check the OpenGL ES version. This sample uses some code from the Compatibility Test Suite (CTS), examining `PackageManager` to determine the major level:

```
// following from
// https://android.googlesource.com/platform/cts/+/master/tests/tests/graphics/src/android/opengl/cts/
```

OpenGlEsVersionTest.java

```
/*
 * Copyright (C) 2010 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in
 * compliance with the License. You may obtain a copy of
 * the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
 * writing, software distributed under the License is
 * distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See
 * the License for the specific language governing
 * permissions and limitations under the License.
 */

private static int getVersionFromPackageManager(Context context) {
  PackageManager packageManager=context.getPackageManager();
  FeatureInfo[] featureInfos=
      packageManager.getSystemAvailableFeatures();
  if (featureInfos != null && featureInfos.length > 0) {
    for (FeatureInfo featureInfo : featureInfos) {
      // Null feature name means this feature is the open
      // gl es version feature.
      if (featureInfo.name == null) {
        if (featureInfo.reqGlEsVersion != FeatureInfo.GL_ES_VERSION_UNDEFINED) {
          return getMajorVersion(featureInfo.reqGlEsVersion);
        }
        else {
          return 1; // Lack of property means OpenGL ES
                    // version 1
        }
      }
    }
  }
  return 1;
}

/** @see FeatureInfo#getGlEsVersion() */
private static int getMajorVersion(int glEsVersion) {
  return((glEsVersion & 0xffff0000) >> 16);
}
```

(from MapsV2/Basic/app/src/main/java/com/commonsware/android/mapsv2/basic/AbstractMapActivity.java)

If the major version is `2` or higher, we return `true` from `readyToGo()`, so `MainActivity` knows to continue on setting up the map. If the major version is `1`, we display a `Toast` — a production-grade app would do something else to let the user know of the problem, most likely.

But, what if `isGooglePlayServicesAvailable()` returns something else?

There are two major possibilities here:

1. The error is something that the user might be able to rectify, such as by downloading the Google Play Services app from the Play Store
2. The error is something that the user cannot recover from

We can distinguish these two cases by calling `isUserResolvableError()` on the `GoogleApiAvailability` singleton, passing in the value we received from `isGooglePlayServicesAvailable()`. This will return `true` if the user might be able to fix the problem, `false` otherwise.

In the `false` case, the user is just out of luck, so we display a `Toast` to alert them of this fact, then `finish()` the activity and return `false`, so `MainActivity` skips over the rest of its work.

In the `true` case, we can display something to the user to prompt them to fix the problem. One way to do that is to use a dialog obtained from Google code, by calling the static `getErrorDialog()` method on a `GoogleApiAvailability` singleton. In our case, we wrap that in a `DialogFragment` named `ErrorDialogFragment`, implemented as a static inner class of `AbstractMapActivity`:

```
public static class ErrorDialogFragment extends DialogFragment {
  static final String ARG_ERROR_CODE="errorCode";

  static ErrorDialogFragment newInstance(int errorCode) {
    Bundle args=new Bundle();
    ErrorDialogFragment result=new ErrorDialogFragment();

    args.putInt(ARG_ERROR_CODE, errorCode);
    result.setArguments(args);

    return(result);
  }

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    Bundle args=getArguments();
    GoogleApiAvailability checker=
      GoogleApiAvailability.getInstance();

    return(checker.getErrorDialog(getActivity(),
      args.getInt(ARG_ERROR_CODE), 0));
  }
```

```
    @Override
    public void onDismiss(DialogInterface dlg) {
      if (getActivity()!=null) {
        getActivity().finish();
      }
    }
  }
```

(from MapsV2/Basic/app/src/main/java/com/commonsware/android/mapsv2/basic/AbstractMapActivity.java)

While the code and comments around `getErrorDialog()` suggest that there is some way for us to find out if the user performed actions that fix the problem, this code does not seem to work well in practice. After all, downloading Google Play Services is asynchronous, so even if the user returns to our app, it is entirely likely that Maps V2 is still unavailable. As a result, when the user is done with the dialog, we `finish()` the activity, forcing the user to start it again if and when they are done downloading Google Play Services.

Testing this code requires an older device, one in which the "Google Play services" app can be uninstalled... if it can be installed at all.

As it turns out, not all Android devices support the Play Store, or the Google Play Services by extension. Notably, if the device lacks the Play Store, `isUserRecoverableError()` returns `true`, even though the user cannot recover from this situation (except perhaps via a firmware update).

(An earlier problem where `getErrorDialog()` could return `null` even for cases where the error is supposedly user-recoverable has been fixed)

Helper Libraries for Maps V2

Many developers have been busy writing libraries that help in the development of Maps V2 applications, beyond Google's own utility library mentioned in the section on animating markers.

Perhaps the most expansive of these is the Android Maps Extensions library. The big thing that this library offers is marker clustering, where as the user zooms out, individual markers are replaced by a marker representing a cluster, so you avoid flooding a small area with too many individual markers:

Figure 612: Map with Many Markers (from Android Maps Extensions demo app)

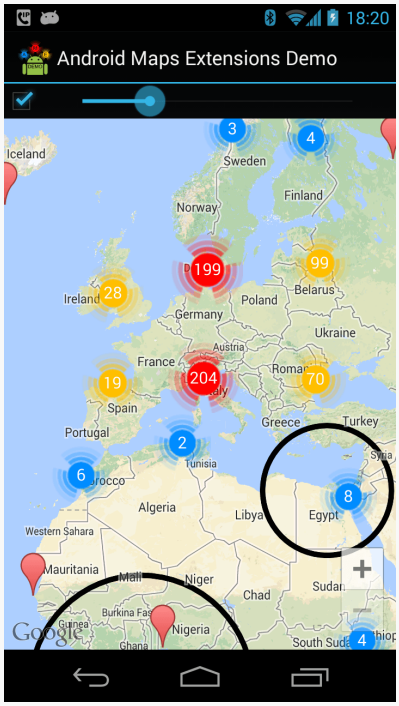


Figure 613: Same Map with Cluster Markers (from Android Maps Extensions demo app)

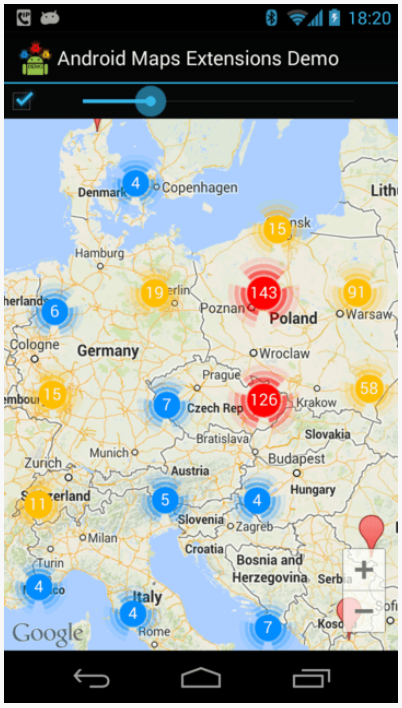


Figure 614: Same Map with Zoomed In Cluster Markers (from Android Maps Extensions demo app)

This library wraps the Maps V2 classes, allowing the library to offer extensions to the standard Maps V2 API, including:

- Associating your own data with `Marker`, `Polygon`, `Polyline`, and other classes, to tie them back to your models
- Getters to retrieve previously-defined markers, etc.
- Etc.

Another library offering marker clustering is clusterkraf, from Two Toasters.

The clusterkraf library can optionally integrate with Cyril Mottier's Polaris2 library. His original Polaris library aimed to provide more features to Maps V1; Polaris2 fills a similar role for Maps V2. At this time, Polaris2 is a smaller library, simply because Maps V2 handles much of what Polaris provided. Polaris2, like Android Maps Extensions, wraps the Maps V2 API with its own classes, in lieu of subclassing (since most Maps V2 classes are marked `final`). Of note, Polaris2 offers `reset()` methods on many of the `...Options` classes (e.g., `MarkerOptions`), and offers constants for the minimum and maximum valid latitude and longitude.

Problems with Maps V2 at Runtime

Portions of the logic that powers your Maps V2 `MapFragment` are supplied by the Google Play Services app. As a result, many operations with Maps V2, such as manipulating markers, require IPC calls between your app and Google Play Services. If those IPC calls are synchronous, they will add a bit of overhead to your app — enough that you will want to avoid them in time-critical pieces of code, tight loops, and the like.

Problems with Maps V2 Deployment

Of course, the key question is: should you be using Maps V2 at all?

Google thinks so, as they have turned off access to new API keys for Maps V1. That makes ongoing development of Maps V1 solutions a bit risky, as you cannot create new API keys for new signing keys, such as if you need to replace your debug keystore.

However, Maps V2 has some deployment limitations at this time. While 99.8+% of Android devices that have the Play Store have the requisite OpenGL ES 2.0+, some devices that have a suitable OpenGL ES version may not have the Play Store or otherwise be unable to get Google Play Services, required for using Maps V2. The `isGooglePlayServicesAvailable()` approach advocated by Google can help determine this at runtime, though this approach used to have some bugs, and it still cannot always help you recover from this problem.

And, as the next section illustrates, not every Android device supports Maps V2, because not every device supports Google Play Services.

What Non-Compliant Devices Show

If your app tries to bring up Maps V2 on a device that cannot possibly have the Play Services Framework — such as a Kindle Fire — the user will see an error dialog:

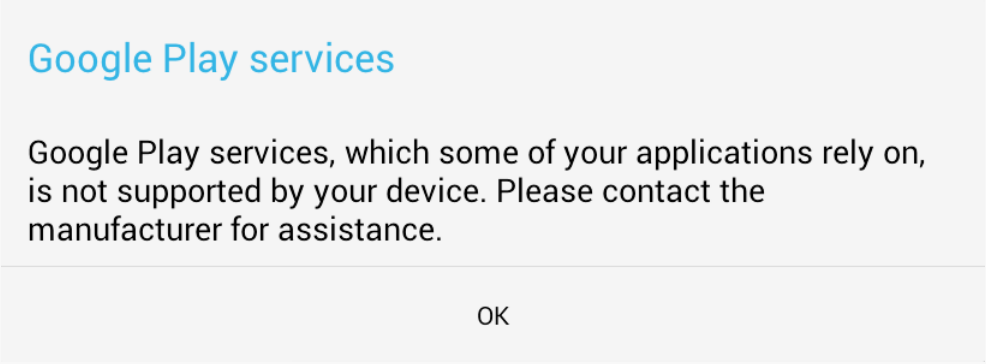


Figure 615: Maps V2 Error on Kindle Fire

For those devices, you will need to consider some alternative source of maps.

Mapping Alternatives

Beyond using Maps V2 or Maps V1, you may need to consider other mapping alternatives. The Google mapping APIs are only available on Android devices that have the Maps SDK add-on (Maps V1) or Google Play Services (Maps V2). Not all devices have those. And, the limitations of Maps V2 deployment and the deprecation of Maps V1 may convince you that relying upon Google for maps is not safe at the present time.

The most common native replacement for Google's mapping is OpenStreetMap, which to some extent is "the Wikipedia of maps". OSMDroid is a library that provides a Maps V1-ish API for embedding OpenStreetMap-based maps into your application.

Another solution is to integrate Web-based Google maps into your app, the same way that you might embed them into your Web site. An activity hosting a `WebView` can display a Web-based Google Map, for example.

Certain devices may have access to other native mapping solutions. For example, Amazon has published their own maps API for use with the Kindle Fire.

Crafting Your Own Views

One of the classic forms of code reuse is the GUI widget. Since the advent of Microsoft Windows — and, to some extent, even earlier – developers have been creating their own widgets to extend an existing widget set. These range from 16-bit Windows "custom controls" to 32-bit Windows OCX components to the innumerable widgets available for Java Swing and SWT, and beyond. Android lets you craft your own widgets as well, such as extending an existing widget with a new UI or new behaviors.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Pick Your Poison

You have five major options for creating a custom `View` class.

First, your "custom `View` class" might really only be custom `Drawable` resources. Many widgets can adopt a radically different look and feel just with replacement graphics. For example, you might think that these toggle buttons from the Android 2.1 Google Maps application are some fancy custom widget:

Figure 616: Google Maps navigation toggle buttons

In reality, those are just radio buttons with replacement images.

Second, your custom `View` class might be a simple subclass of an existing widget, where you override some behaviors or otherwise inject your own logic. Unfortunately, most of the built-in Android widgets are not really designed for this sort of simple subclassing, so you may be disappointed in how well this particular technique works.

Third, your custom `View` class might be a composite widget — akin to an activity's contents, complete with layout and such, but encapsulated in its own class. This allows you to create something more elaborate than you will just by tweaking resources. We will see this later in the chapter with `ColorMixer`.

Fourth, you might want to implement your own layout manager, if your GUI rules do not fit well with `RelativeLayout`, `TableLayout`, or other built-in containers. For example, you might want to create a layout manager that more closely mirrors the "box model" approach taken by XUL and Flex, or you might want to create one that mirrors Swing's `FlowLayout` (laying widgets out horizontally until there is no more room on the current row, then start a new row).

Finally, you might want to do something totally different, where you need to draw the widget yourself. For example, the `ColorMixer` widget uses `SeekBar` widgets to control the mix of red, blue, and green. But, you might create a `ColorWheel` widget that draws a spectrum gradient, detects touch events, and lets the user pick a color that way.

Some of these techniques are fairly simple; others are fairly complex. All share some common traits, such as widget-defined attributes, that we will see throughout the remainder of this chapter.

Colors, Mixed How You Like Them

The classic way for a user to pick a color in a GUI is to use a color wheel like this one:

Figure 617: A color wheel from the API samples

There is even code to make one in the API samples.

However, a color wheel like that is difficult to manipulate on a touch screen, particularly a capacitive touchscreen designed for finger input. Fingers are great for gross touch events and lousy for selecting a particular color pixel.

Another approach is to use a mixer, with sliders to control the red, green, and blue values:

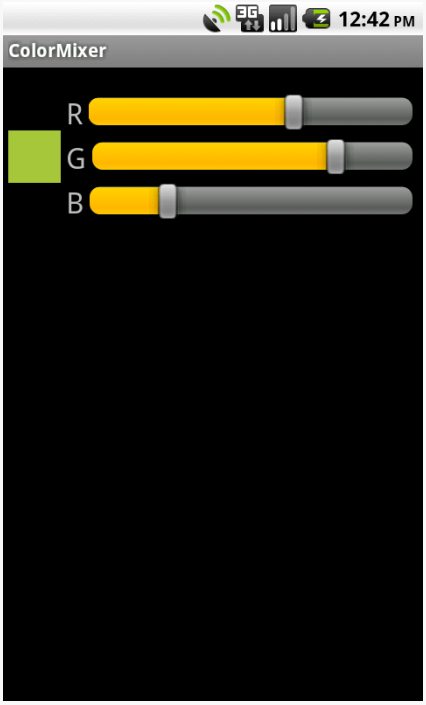


Figure 618: The ColorMixer widget, inside an activity

That is the custom widget you will see in this section, based on the code in the Views/ColorMixer sample project.

The Layout

`ColorMixer` is a composite widget, meaning that its contents are created from other widgets and containers. Hence, we can use a layout file to describe what the widget should look like.

The layout to be used for the widget is not that much: three `SeekBar` widgets (to control the colors), three `TextView` widgets (to label the colors), and one plain `View` (the "swatch" on the left that shows what the currently selected color is). Here is the file, found in `res/layout/mixer.xml` in the `Views/ColorMixer` project:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <View android:id="@+id/swatch"
    android:layout_width="40dip"
    android:layout_height="40dip"
    android:layout_alignParentLeft="true"
    android:layout_centerVertical="true"
```

```
    android:layout_marginLeft="4dip"
  />
  <TextView android:id="@+id/redLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/swatch"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:text="@string/red"
    android:textSize="24sp"
  />
  <SeekBar android:id="@+id/red"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/redLabel"
    android:layout_toRightOf="@id/redLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
  <TextView android:id="@+id/greenLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/redLabel"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/green"
    android:textSize="24sp"
  />
  <SeekBar android:id="@+id/green"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/greenLabel"
    android:layout_toRightOf="@id/greenLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
  <TextView android:id="@+id/blueLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/greenLabel"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/blue"
    android:textSize="24sp"
  />
  <SeekBar android:id="@+id/blue"
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/blueLabel"
    android:layout_toRightOf="@id/blueLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
</merge>
```

(from Views/ColorMixer/app/src/main/res/layout/mixer.xml)

One thing that is a bit interesting about this layout, though, is the root element: `<merge>`. A `<merge>` layout is a bag of widgets that can be poured into some other container. The layout rules on the children of `<merge>` are then used in conjunction with whatever container they are added to. As we will see shortly, `ColorMixer` itself inherits from `RelativeLayout`, and the children of the `<merge>` element will become children of `ColorMixer` in Java. Basically, the `<merge>` element is only there because XML files need a single root — otherwise, the `<merge>` element itself is ignored in the layout.

The Attributes

Widgets usually have attributes that you can set in the XML file, such as the `android:src` attribute you can specify on an `ImageButton` widget. You can create your own custom attributes that can be used in your custom widget, by creating a `res/values/attrs.xml` file containing `declare-styleable` resources to specify them.

For example, here is the attributes file for `ColorMixer`:

```
<resources>
  <declare-styleable name="ColorMixer">
    <attr name="initialColor" format="color" />
  </declare-styleable>
</resources>
```

(from Views/ColorMixer/app/src/main/res/values/attrs.xml)

The `declare-styleable` element describes what attributes are available on the widget class specified in the name attribute — in our case, `ColorMixer`. Inside `declare-styleable` you can have one or more `attr` elements, each indicating the `name` of an attribute (e.g., `initialColor`) and what data `format` the attribute has (e.g., `color`). The data type will help with compile-time validation and in getting any supplied values for this attribute parsed into the appropriate type at runtime.

Here, we indicate there is only one attribute: `initialColor`, which will hold the initial color we want the mixer set to when it first appears.

There are many possible values for the `format` attribute in an `attr` element, including:

1. `boolean`
2. `color`
3. `dimension`
4. `float`
5. `fraction`
6. `integer`
7. `reference` (which means a reference to another resource, such as a `Drawable`)
8. `string`

You can even support multiple formats for an attribute, by separating the values with a pipe (e.g., `reference|color`).

The Class

Our `ColorMixer` class, a subclass of `RelativeLayout`, will take those attributes and provide the actual custom widget implementation, for use in activities.

Constructor Flavors

A `View` has three possible constructors:

1. One takes just a `Context`, which usually will be an `Activity`
2. One takes a `Context` and an `AttributeSet`, the latter of which represents the attributes supplied via layout XML
3. One takes a `Context`, an `AttributeSet`, and the default style to apply to the attributes

If you are expecting to use your custom widget in layout XML files, you will need to implement the second constructor and chain to the superclass. If you want to use styles with your custom widget when declared in layout XML files, you will need to implement the third constructor and chain to the superclass. If you want developers to create instances of your `View` class in Java code directly, you probably should implement the first constructor and, again, chain to the superclass.

In the case of `ColorMixer`, all three constructors are implemented, eventually routing to the three-parameter edition, which initializes our widget. Below, you will see the first two of those constructors, with the third coming up in the next section:

```
public ColorMixer(Context context) {
  this(context, null);
}

public ColorMixer(Context context, AttributeSet attrs) {
  this(context, attrs, 0);
}
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

Using the Attributes

The `ColorMixer` has a starting color — after all, the `SeekBar` widgets and swatch `View` have to show something. Developers can, if they wish, set that color via a `setColor()` method:

```
public void setColor(int color) {
  red.setProgress(Color.red(color));
  green.setProgress(Color.green(color));
  blue.setProgress(Color.blue(color));
  swatch.setBackgroundColor(color);
}
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

If, however, we want developers to be able to use layout XML, we need to get the value of `initialColor` out of the supplied `AttributeSet`. In `ColorMixer`, this is handled in the three-parameter constructor:

```
public ColorMixer(Context context, AttributeSet attrs, int defStyle) {
  super(context, attrs, defStyle);

  ((Activity)getContext())
      .getLayoutInflater()
      .inflate(R.layout.mixer, this, true);

  swatch=findViewById(R.id.swatch);

  red=(SeekBar)findViewById(R.id.red);
  red.setMax(0xFF);
  red.setOnSeekBarChangeListener(onMix);
```

```
    green=(SeekBar)findViewById(R.id.green);
    green.setMax(0xFF);
    green.setOnSeekBarChangeListener(onMix);

    blue=(SeekBar)findViewById(R.id.blue);
    blue.setMax(0xFF);
    blue.setOnSeekBarChangeListener(onMix);

    if (attrs!=null) {
      TypedArray a=getContext()
                      .obtainStyledAttributes(attrs,
                                              R.styleable.ColorMixer,
                                              0, 0);

      setColor(a.getInt(R.styleable.ColorMixer_initialColor,
                        0xFFA4C639));
      a.recycle();
    }
  }
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

There are three steps for getting attribute values:

- Get a `TypedArray` conversion of the `AttributeSet` by calling `obtainStyledAttributes()` on our `Context`, supplying it the `AttributeSet` and the ID of our styleable resource (in this case, `R.styleable.ColorMixer`, since we set the name of the `declare-styleable` element to be `ColorMixer`)
- Use the `TypedArray` to access specific attributes of interest, by calling an appropriate getter (e.g., `getInt()`) with the ID of the specific attribute to fetch (`R.styleable.ColorMixer_initialColor`)
- Recycle the `TypedArray` when done, via a call to `recycle()`, to make the object available to Android for use with other widgets via an object pool (versus creating new instances every time)

Note that the name of any given attribute, from the standpoint of `TypedArray`, is the name of the styleable resource (`R.styleable.ColorMixer`) concatenated with an underscore and the name of the attribute itself (`_initialColor`).

In `ColorMixer`, we get the attribute and pass it to `setColor()`. Since `getInt()` on `AttributeSet` takes a default value, we supply some stock color that will be used if the developer declined to supply an `initialColor` attribute.

Also note that our `ColorMixer` constructor inflates the widget's layout. In particular, it supplies `true` as the third parameter to `inflate()`, meaning that the contents of the layout should be added as children to the `ColorMixer` itself. When the layout is inflated, the `<merge>` element is ignored, and the `<merge>` element's children are added as children to the `ColorMixer`.

Saving the State

Similar to activities, a custom `View` overrides `onSaveInstanceState()` and `onRestoreInstanceState()` to persist data as needed, such as to handle a screen orientation change. The biggest difference is that rather than receive a `Bundle` as a parameter, `onSaveInstanceState()` must return a `Parcelable` with its state... including whatever state comes from the parent `View`.

The simplest way to do that is to return a `Bundle`, in which we have filled in our state (the chosen color) and the parent class' state (whatever that may be).

So, for example, here are implementations of `onSaveInstanceState()` and `onRestoreInstanceState()` from `ColorMixer`:

```
@Override
public Parcelable onSaveInstanceState() {
  Bundle state=new Bundle();

  state.putParcelable(SUPERSTATE, super.onSaveInstanceState());
  state.putInt(COLOR, getColor());

  return(state);
}

@Override
public void onRestoreInstanceState(Parcelable ss) {
  Bundle state=(Bundle)ss;

  super.onRestoreInstanceState(state.getParcelable(SUPERSTATE));

  setColor(state.getInt(COLOR));
}
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

The Rest of the Functionality

`ColorMixer` defines a callback interface, named `OnColorChangedListener`:

```
public interface OnColorChangedListener {
  public void onColorChange(int argb);
}
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

`ColorMixer` also provides getters and setters for an `OnColorChangedListener` object:

```
public OnColorChangedListener getOnColorChangedListener() {
  return(listener);
}

public void setOnColorChangedListener(OnColorChangedListener listener) {
  this.listener=listener;
}
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

The rest of the logic is mostly tied up in the `SeekBar` handler, which will adjust the swatch based on the new color and invoke the `OnColorChangedListener` object, if there is one:

```
private SeekBar.OnSeekBarChangeListener onMix=new SeekBar.OnSeekBarChangeListener() {
  public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
    int color=getColor();

    swatch.setBackgroundColor(color);

    if (listener!=null) {
      listener.onColorChange(color);
    }
  }

  public void onStartTrackingTouch(SeekBar seekBar) {
    // unused
  }

  public void onStopTrackingTouch(SeekBar seekBar) {
    // unused
  }
};
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixer.java)

Seeing It In Use

The project contains a sample activity, `ColorMixerDemo`, that shows the use of the `ColorMixer` widget.

The layout for that activity, shown below, can be found in `res/layout/main.xml` of the `Views/ColorMixer` project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:mixer="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
>
  <TextView android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <com.commonsware.android.colormixer.ColorMixer
    android:id="@+id/mixer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    mixer:initialColor="#FFA4C639"
  />
</LinearLayout>
```

(from Views/ColorMixer/app/src/main/res/layout/main.xml)

Notice that the root `LinearLayout` element defines two namespaces, the standard `android` namespace, and a separate one named `mixer`. The `mixer` namespace is given a URL of `http://schemas.android.com/apk/res-auto`, which indicates to the Android build system to match up `mixer` attributes with their respective widgets that are supplied via Android library projects.

Our `ColorMixer` widget is in the layout, with a fully-qualified class name (`com.commonsware.android.colormixer.ColorMixer`), since `ColorMixer` is not in the `android.widget` package. Notice that we can treat our custom widget like any other, giving it a width and height and so on.

The one attribute of our `ColorMixer` widget that is unusual is `mixer:initialColor`. `initialColor`, you may recall, was the name of the attribute we declared in `res/values/attrs.xml` and retrieve in Java code, to represent the color to start with. The `mixer` namespace is needed to identify where Android should be pulling the rules for what sort of values an `initialColor` attribute can hold. Since our `<attr>` element indicated that the `format` of `initialColor` was `color`, Android will expect to see a color value here, rather than a string or dimension.

The `ColorMixerDemo` activity is not very elaborate:

```
package com.commonsware.android.colormixer;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class ColorMixerDemo extends Activity {
  private TextView color=null;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    color=(TextView)findViewById(R.id.color);

    ColorMixer mixer=(ColorMixer)findViewById(R.id.mixer);

    mixer.setOnColorChangedListener(onColorChange);
  }

  private ColorMixer.OnColorChangedListener onColorChange=
    new ColorMixer.OnColorChangedListener() {
    public void onColorChange(int argb) {
      color.setText(Integer.toHexString(argb));
    }
  };
}
```

(from Views/ColorMixer/app/src/main/java/com/commonsware/android/colormixer/ColorMixerDemo.java)

It gets access to both the `ColorMixer` and the `TextView` in the main layout, then registers an `OnColorChangedListener` with the `ColorMixer`. That listener, in turn, puts the value of the color in the `TextView`, so the user can see the hex value of the color along with the shade itself in the swatch.

ReverseChronometer: Simply a Custom Subclass

Sometimes, what you want to achieve only requires a basic subclass of an existing widget (or container), into which you can pour your business logic.

For example, Android has a `Chronometer` widget, which is used for denoting elapsed time of some operation. It works well, but it only counts up from zero. It cannot be used to display a countdown instead.

But, we can roll a `ReverseChronometer` that does, simply by subclassing `TextView`, as seen in the Views/ReverseChronometer sample project:

```
package com.commonsware.android.revchron;

import android.content.Context;
import android.graphics.Color;
import android.os.SystemClock;
import android.util.AttributeSet;
import android.widget.TextView;

public class ReverseChronometer extends TextView implements Runnable {
  long startTime=0L;
  long overallDuration=0L;
  long warningDuration=0L;

  public ReverseChronometer(Context context, AttributeSet attrs) {
    super(context, attrs);

    reset();
  }

  @Override
  public void run() {
    long elapsedSeconds=
        (SystemClock.elapsedRealtime() - startTime) / 1000;

    if (elapsedSeconds < overallDuration) {
      long remainingSeconds=overallDuration - elapsedSeconds;
      long minutes=remainingSeconds / 60;
      long seconds=remainingSeconds - (60 * minutes);

      setText(String.format("%d:%02d", minutes, seconds));

      if (warningDuration > 0 && remainingSeconds < warningDuration) {
        setTextColor(0xFFFF6600); // orange
      }
      else {
        setTextColor(Color.BLACK);
      }

      postDelayed(this, 1000);
    }
    else {
      setText("0:00");
      setTextColor(Color.RED);
    }
```

```
  }

  public void reset() {
    startTime=SystemClock.elapsedRealtime();
    setText("--:--");
    setTextColor(Color.BLACK);
  }

  public void stop() {
    removeCallbacks(this);
  }

  public void setOverallDuration(long overallDuration) {
    this.overallDuration=overallDuration;
  }

  public void setWarningDuration(long warningDuration) {
    this.warningDuration=warningDuration;
  }
}
```

(from Views/ReverseChronometer/app/src/main/java/com/commonsware/android/revchron/ReverseChronometer.java)

`ReverseChronometer` is designed to show minutes and seconds remaining from some initial time. In the constructor, by means to a call to a `reset()` method, we set the text of the `TextView` to show a generic starting point ("-:–"), set its color to black, and note the current time (`SystemClock.elapsedRealtime()`) in a `startTime` data member.

`ReverseChronometer` also tracks two durations in seconds, with corresponding setter methods:

- `overallDuration` is how long the countdown should run from beginning to end
- `warningDuration` is how far from the end we should change the color of the `TextView` from black to orange, to hint to the viewer that time is running out

`ReverseChronometer` implements `Runnable`, and when its `run()` method is called, it determines how many seconds have elapsed since that `startTime` value. Depending on the amount of seconds remaining, we either:

- Just update the text to show the minutes and seconds remaining
- Update the text and set the color to black or orange
- Set the text to "0:00" (time has run out) and set the text color to red

In either of the first two cases, we also call `postDelayed()` to schedule ourselves to run again in a second, where we can update the `TextView` contents once more. That continues until somebody calls `stop()`.

As with any custom `View`, we can reference this in a layout XML resource, fully-qualifying the class name used as the name of our XML element for the widget. And, since we inherit from `TextView`, we can set any of the attributes that we want on that `TextView`, in terms of styling the text, positioning it within a parent container, etc.:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">

  <com.commonsware.android.revchron.ReverseChronometer
    android:id="@+id/chrono"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:textSize="50sp"
    android:textStyle="bold"/>

</RelativeLayout>
```

(from Views/ReverseChronometer/app/src/main/res/layout/activity_main.xml)

All our activity needs to do is set the durations, then call `run()` and `stop()` at appropriate times, such as when the activity is resumed and paused:

```
package com.commonsware.android.revchron;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
  private ReverseChronometer chrono=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    chrono=(ReverseChronometer)findViewById(R.id.chrono);
    chrono.setOverallDuration(90);
    chrono.setWarningDuration(10);
```

```
  }

  @Override
  public void onResume() {
    super.onResume();

    chrono.run();
  }

  @Override
  public void onPause() {
    chrono.stop();

    super.onPause();
  }
}
```

(from Views/ReverseChronometer/app/src/main/java/com/commonsware/android/revchron/MainActivity.java)

The result is much as you would expect: a countdown of the time remaining:

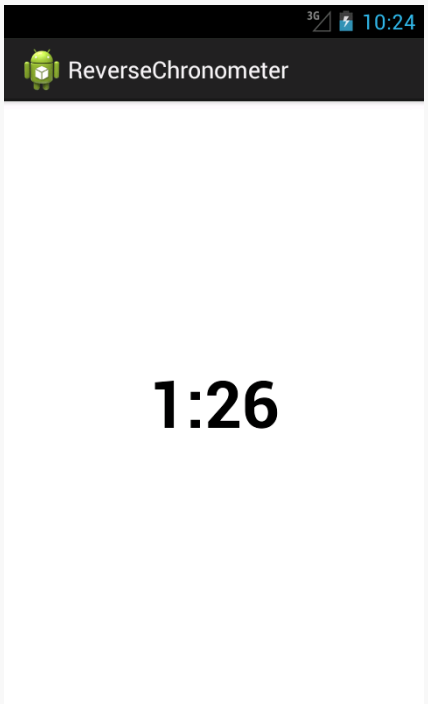


Figure 619: ReverseChronometer, Early in Countdown

...changing to orange when we are within the warning duration:

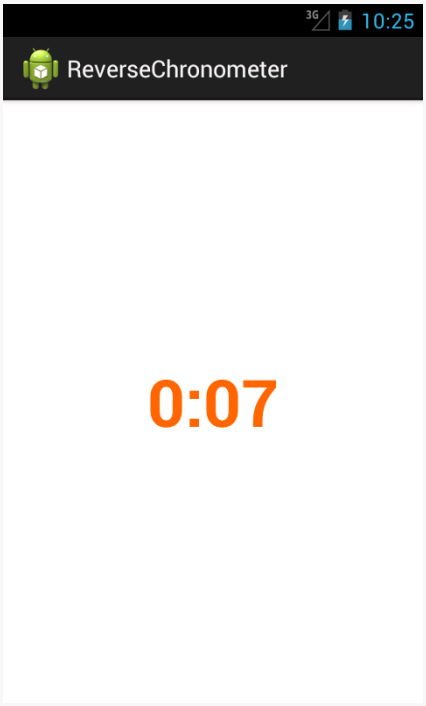


Figure 620: ReverseChronometer, Late in Countdown

...and changing to red when time has run out:

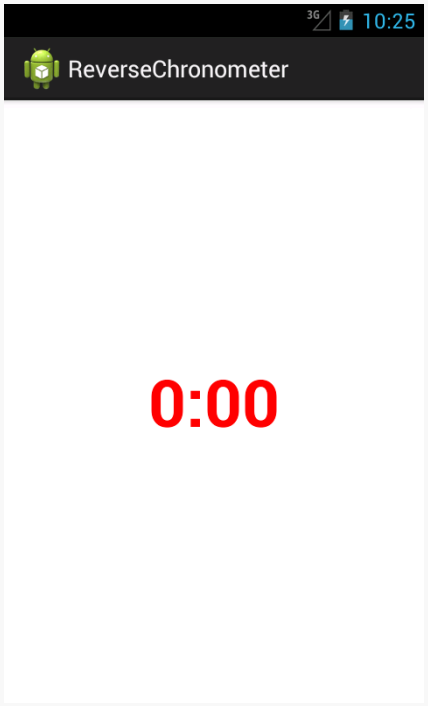


Figure 621: ReverseChronometer, With Complete Time Elapsed

Of course, much more could be done with this widget, if you chose:

- Support other constructors, beyond the two-argument constructor needed for layout inflation
- Support setting durations and colors via custom XML attributes
- Adding listeners for warning and expired events, so other things can be done at those points in time (e.g., play a sound, vibrate the device)

AspectLockedFrameLayout: A Custom Container

You can also craft your own custom container classes, whether inheriting straight from `ViewGroup` to implement your own set of layout rules, or by extending an existing `ViewGroup` to merely augment its functionality.

For example, there may be cases where you want to control the aspect ratio of some set of widgets. This is important when working with preview frames off of the `Camera` to prevent distortion, for example.

`AspectLockedFrameLayout`, therefore, is a custom extension of `FrameLayout` that ensures that its contents are kept within a particular aspect ratio, reducing the height or width of the contents to keep that aspect ratio.

`AspectLockedFrameLayout` is published as part of the CWAC-Layouts project, with its own GitHub repo. As with many of the CWAC projects, the reusable code is distributed as a JAR and as an Android library project, with a `demo/` sub-project illustrating the use of some of the library's contents.

`AspectLockedFrameLayout` holds onto two data members:

- A `double` (`aspectRatio`) that represents a specific aspect ratio to maintain, initialized to `0.0`
- A `View` (`aspectRatioSource`) that represents some other widget whose aspect ratio should be matched, initialized to `null`

`AspectLockedFrameLayout` has corresponding setters for each:

```
      lockedHeight=(int)(lockedWidth / localRatio + .5);
    }

    // Add the padding of the border.
    lockedWidth+=hPadding;
    lockedHeight+=vPadding;

    // Ask children to follow the new preview dimension.
    super.onMeasure(MeasureSpec.makeMeasureSpec(lockedWidth,
                                                MeasureSpec.EXACTLY),
                    MeasureSpec.makeMeasureSpec(lockedHeight,
                                                MeasureSpec.EXACTLY));
  }
}

/**
 * Supplies a View as a source. The AspectLockedFrameLayout will aim to
 * match the aspect ratio of this View. This is a one-time check; if the
 * View changes its aspect ratio later, the AspectLockedFrameLayout will
 * not attempt to match it.
 *
 * @param v some View
```

The "business logic" of maintaining the aspect ratio comes in `onMeasure()`. `onMeasure()` is called on a `ViewGroup` when it is time for it to determine its actual size, based upon things like the requested height and width and the sizes of its

children. In our case `onMeasure()` needs to be tweaked to maintain the aspect ratio, assuming that we have an aspect ratio to work with:

```
  }

  /**
   * {@inheritDoc}
   */
  public AspectLockedFrameLayout(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  // from com.android.camera.PreviewFrameLayout, with slight
  // modifications

  /**
   * {@inheritDoc}
   */
  @Override
  protected void onMeasure(int widthSpec, int heightSpec) {
    double localRatio=aspectRatio;

    if (localRatio == 0.0 && aspectRatioSource != null
        && aspectRatioSource.getHeight() > 0) {
      localRatio=
          (double)aspectRatioSource.getWidth()
              / (double)aspectRatioSource.getHeight();
    }

    if (localRatio == 0.0) {
      super.onMeasure(widthSpec, heightSpec);
    }
    else {
      int lockedWidth=MeasureSpec.getSize(widthSpec);
      int lockedHeight=MeasureSpec.getSize(heightSpec);

      if (lockedWidth == 0 && lockedHeight == 0) {
        throw new IllegalArgumentException(
                                          "Both width and height cannot be zero -- watch out for
scrollable containers");
      }

      // Get the padding of the border background.
      int hPadding=getPaddingLeft() + getPaddingRight();
      int vPadding=getPaddingTop() + getPaddingBottom();

      // Resize the preview frame with correct aspect ratio.
      lockedWidth-=hPadding;
      lockedHeight-=vPadding;

      if (lockedHeight > 0 && (lockedWidth > lockedHeight * localRatio)) {
        lockedWidth=(int)(lockedHeight * localRatio + .5);
      }
```

We start by determining what actually is the desired aspect ratio, held onto in a `localRatio` local variable. That will be `aspectRatio` if we do not have an `aspectRatioSource` that already knows its size, otherwise we will calculate the

aspect ratio from the source. And, if `localRatio` turns out to be `0.0`, indicating that we do not have an aspect ratio to maintain, we just chain to the superclass, so `AspectLockedFrameLayout` will behave just like a normal `FrameLayout`.

If we *do* have an aspect ratio to maintain, we start by determining our requested height and width. `onMeasure()` is passed a pair of "specs" that provides details about our requested size, and we can get the height and width from those by means of the `MeasureSpec` helper class. We remove any horizontal padding — padding is considered to be "outside" the locked area and therefore is ignored in aspect ratio calculations. We then adjust the height or the width, as needed, to maintain the aspect ratio. We add back in the padding, then chain to the superclass with revised height and width "specs" via `MeasureSpec`.

Note that much of this logic was derived from `com.android.camera.PreviewFrameLayout` from the AOSP Camera application, which is used to maintain the aspect ratio of the `SurfaceView` used to display preview frames.

To use an `AspectLockedFrameLayout`, just add it to your layout XML file, with an appropriate child widget/container representing the material that needs to maintain a particular aspect ratio. Since the `AspectLockedFrameLayout` is overriding its natural size, you can use `android:layout_gravity` to control its positioning within some parent widget, such as centering it:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.commonsware.cwac.layouts.AspectLockedFrameLayout
        android:id="@+id/source"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center">

        <!-- children go here -->
    </com.commonsware.cwac.layouts.AspectLockedFrameLayout>
</FrameLayout>
```

Mirror and MirroringFrameLayout: Draw It Yourself

Another scenario where aspect ratios matter is when you are presenting information on an external display via `Presentation`, as is covered elsewhere in this book.

Ideally, you fill the external display. And normally this will happen for you automatically, as your `Presentation` content view should fill the available screen space... assuming that the content has the right aspect ratio, or can be suitably stretched.

One scenario where this might be a problem is if you want *the same material* shown on both the main display and on the external display. For example, suppose that you are using `Presentation` to deliver... well... a presentation. The external display is probably some form of video projector, and you will want your slides or other materials shown there. However, it is useful for you to be able to see those same slides and such on the tablet, as typically the projector screen is behind, or to the side of, the presenter. If the presenter has to keep turning around to confirm what is shown on "the big screen", it can detract from the presentation.

Moreover, you might not only want to *show* the same material, but have it stem *from the same source*, on the tablet, for interactivity reasons. Suppose that you want to display a Web page. You might just pop up a `WebView` in the `Presentation`. But... how do you scroll? The `Presentation` offers no touch interface — projector screens do not magically respond to pinch-to-zoom just because we happen to be projecting something onto them from an Android tablet.

In this case, ideally we would like to *mirror* something. Have the actual widgets shown on the tablet, which can then respond to touch events and the like. At the same time, capture what is shown on the tablet and reproduce it, verbatim, on the `Presentation` for the audience to see. Now everybody can see the same material, and the presenter can manipulate that material.

But now aspect ratios come into play. We want to fill the `Presentation` display space, without black bars or stretching or whatever. That only works if our source material — the widgets and containers to be mirrored — have the same aspect ratio as the `Presentation`'s `Display` itself.

With that in mind, the CWAC Layouts project also contains two classes to solve this problem:

- `MirroringFrameLayout` is an `AspectLockedFrameLayout` that also can mirror its content to...
- `Mirror`, a `View` that takes a `Bitmap` representing the `MirroringFrameLayout` contents and displays it

Technically, `MirroringFrameLayout` works with a `MirrorSink`, an interface that can receive updates to the content to be mirrored when that content changes. `Mirror` implements `MirrorSink`, and you could have other classes implement `MirrorSink` as well if that made sense for your app. The sections that follow focus on `MirroringFrameLayout` working with a `Mirror`, as that is the most likely scenario.

MirroringFrameLayout

`MirroringFrameLayout` extends `AspectLockedFrameLayout`, so that we can lock the aspect ratio of the to-be-mirrored contents to match the aspect ratio of the `Mirror`. The `Mirror` is designed to be projected by the `Presentation`, and so if the `Mirror` fills the `Presentation`'s `Display`, we want our `MirroringFrameLayout` to match the aspect ratio so the entire `Display` can indeed be filled.

Of course, a `ViewGroup` like `FrameLayout` normally just has its children draw to the screen. In our case, we need to capture what is drawn ourselves, to supply to the `Mirror` as needed. This is a bit tricky.

```
package com.commonsware.cwac.layouts;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.util.AttributeSet;
import android.view.ViewTreeObserver.OnPreDrawListener;
import android.view.ViewTreeObserver.OnScrollChangedListener;

/**
 * A FrameLayout that locks its aspect ratio (courtesy of AspectLockedFrameLayout)
 * and supplies "screenshots" of its contents to an associated MirrorSink,
 * such as a Mirror.
 *
 * Principally, MirroringFrameLayout and Mirror are designed for use with
 * Android's Presentation system. The MirroringFrameLayout would be part of the
 * UI of the activity on the mobile device, allowing for user interaction. The
 * Mirror would be used in the Presentation to show an audience (e.g., via a
 * projector) what is shown inside the MirroringFrameLayout on the mobile
 * device.
 */
public class MirroringFrameLayout extends AspectLockedFrameLayout
    implements OnPreDrawListener, OnScrollChangedListener {
  private MirrorSink mirror=null;
  private Bitmap bmp=null;
  private Canvas bmpBackedCanvas=null;
  private Rect rect=new Rect();

  /**
   * {@inheritDoc}
   */
  public MirroringFrameLayout(Context context) {
```

```
  this(context, null);
}

/**
 * {@inheritDoc}
 */
public MirroringFrameLayout(Context context, AttributeSet attrs) {
  super(context, attrs);

  setWillNotDraw(false);
}

/**
 * Associate a MirrorSink; this sink will be given bitmaps representing
 * updated contents of the MirroringFrameLayout as those contents change.
 *
 * @param mirror a Mirror or other MirrorSink implementation
 */
public void setMirror(MirrorSink mirror) {
  this.mirror=mirror;

  if (mirror != null) {
    setAspectRatioSource(mirror);
  }
}

/**
 * {@inheritDoc}
 */
@Override
public void onAttachedToWindow() {
  super.onAttachedToWindow();

  getViewTreeObserver().addOnPreDrawListener(this);
  getViewTreeObserver().addOnScrollChangedListener(this);
}

/**
 * {@inheritDoc}
 */
@Override
public void onDetachedFromWindow() {
  getViewTreeObserver().removeOnPreDrawListener(this);
  getViewTreeObserver().removeOnScrollChangedListener(this);

  super.onDetachedFromWindow();
}

/**
 * {@inheritDoc}
 */
@Override
public void draw(Canvas canvas) {
  if (mirror != null) {
    bmp.eraseColor(0);

    super.draw(bmpBackedCanvas);
    getDrawingRect(rect);
    canvas.drawBitmap(bmp, null, rect, null);
    mirror.update(bmp);
```

```
    }
    else {
      super.draw(canvas);
    }
  }

  /**
   * {@inheritDoc}
   */
  @Override
  protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    initBitmap(w, h);

    super.onSizeChanged(w, h, oldw, oldh);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public boolean onPreDraw() {
    if (mirror != null) {
      if (bmp == null) {
        requestLayout();
      }
      else {
        invalidate();
      }
    }

    return(true);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public void onScrollChanged() {
    onPreDraw();
  }

  private void initBitmap(int w, int h) {
    if (mirror != null) {
      if (bmp == null || bmp.getWidth() != w || bmp.getHeight() != h) {
        if (bmp != null) {
          bmp.recycle();
        }

        bmp=Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);
        bmpBackedCanvas=new Canvas(bmp);
      }
    }
  }
}
```

Our one-argument constructor uses `this()` to chain to the two-argument constructor. The two-argument constructor calls `setWillNotDraw(false)` indicating to Android that we want this `ViewGroup` to participate in the drawing process like a

regular `View` — normally, certain steps in the drawing process are skipped as being irrelevant to `View` classes that do not draw anything themselves.

We have a `setMirror()` method, where the activity or fragment can supply the `MirrorSink` that is connected to this `MirroringFrameLayout`. In addition to holding onto the `MirrorSink` in a `mirror` data member, we call `setAspectRatioSource()`, inherited from `AspectLockedFrameLayout`, so our contents will match the aspect ratio from that source.

`MirroringFrameLayout` overrides `onAttachedToWindow()` and `onDetatchedFromWindow()`. As one might guess, these callbacks are called when views are attached and detached from some window. Usually, that window represents an activity, though it could represent a `Dialog` or a `Presentation`.

In those callbacks, we connect with the `ViewTreeObserver` of the `MirroringFrameLayout`. A `ViewTreeObserver` is a way to find out about events of a view tree, rooted at some `ViewGroup`. In our case, we want to find out when children are going to be drawn (`addOnPreDrawListener()`) and when they are scrolled (`addOnScrollChangedListener()`).

We override `onSizeChanged()`. This is called on any `View` when its size may have changed, either because it is being sized initially when the UI is being set up, or because something else nearby changed size (e.g., its parent) and therefore the size of the `View` itself may now be different. In our case, we use `onSizeChanged()` to set up a `Bitmap` object, sized to match our size, and a `Canvas` object that wraps around that `Bitmap` object. As you will see, we will use this `Canvas` to capture what is being drawn on the screen, for later use by the `Mirror`.

We also override `draw()`. This is, in effect, the "entry point" into the logic that causes a `View` to render itself on the screen, by drawing to a supplied `Canvas` object. Most `View` classes do not override `draw()`, as the real rendering is done in an `onDraw()` method, as we will see with `Mirror` later in this chapter. However, in our case, we have to override `draw()` for one simple reason: we do not want to draw to the `Canvas` supplied by Android to the `draw()` method. We want to draw to our own `Canvas`, backed by that `Bitmap`.

To that end, if we have a `MirrorSink`, we:

- Make sure the `Bitmap` starts off blank by calling `eraseColor()`
- Chain to the superclass, replacing the `Canvas` given to us in `draw()` by our own `Bitmap`-backed `Canvas`

- Calculate a `Rect` object with our size and position, using `getDrawingRect()`
- Use that `Rect` and the `Bitmap` to render the `Bitmap` to the "real" `Canvas` supplied to us in `draw()`
- Call `update()` on the `MirrorSink`, to give it the new `Bitmap`

By rendering our contents to the `Bitmap`-backed `Canvas`, instead of the normal one, we capture a copy of the output, in the form of the `Bitmap`. Since the `Bitmap` has the same size as the "real" `Canvas` (courtesy of our `onSizeChanged()` work), when we draw the `Bitmap` onto the `Canvas`, we effectively "color in" the same pixels in the same spots as if we had skipped all of this and left the normal `draw()` logic alone. But, since we still hold onto our `Bitmap`, we can use those same pixels elsewhere... such as in our `Mirror`.

The problem with relying on `draw()` is that it is not always called when there are changes to widgets within the `MirroringFrameLayout`. In particular, `WebView` often does not trigger `draw()` on the `MirroringFrameLayout`. That's where the pre-draw and scroll-changed events from the `ViewTreeObserver` come into play: they give us more indication that we need to update our `Bitmap`.

The `onPreDraw()` method is called when a child of this `MirroringFrameLayout` is about to be drawn. If we have our `MirrorSink`, we then either call `requestLayout()` (if we have no bitmap yet) or `invalidate()` (if we do), to trigger Android to go through the draw process for the `MirroringFrameLayout` too, allowing us to update our `Bitmap`.

The `onScrollChanged()` method is called when a child of this `MirroringFrameLayout` has been scrolled. This delegates to `onPreDraw()`, to run through the same logic to force an update to the `Bitmap`.

Mirror

`Mirror` extends the base `View` class, and so it is the most "raw" of all the custom widgets and containers shown so far in this chapter. It has an `update()` method, used to connect the `MirroringFrameLayout` from which the `Mirror` can obtain what it is supposed to display:

```
package com.commonsware.cwac.layouts;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Rect;
```

```
import android.util.AttributeSet;
import android.view.View;

/**
 * A View that implements MirrorSink and renders the supplied bitmaps to its
 * own contents. When connected to a MirroringFrameLayout, Mirror will aim to
 * show the same contents as is in the MirroringFrameLayout, at the same aspect
 * ratio, though possibly at a different size.
 *
 * Principally, MirroringFrameLayout and Mirror are designed for use with
 * Android's Presentation system. The MirroringFrameLayout would be part of the
 * UI of the activity on the mobile device, allowing for user interaction. The
 * Mirror would be used in the Presentation to show an audience (e.g., via a
 * projector) what is shown inside the MirroringFrameLayout on the mobile
 * device.
 */
public class Mirror extends View implements MirrorSink {
  private Rect rect=new Rect();
  private Bitmap bmp=null;

  /**
   * {@inheritDoc}
   */
  public Mirror(Context context) {
    super(context);
  }

  /**
   * {@inheritDoc}
   */
  public Mirror(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  /**
   * {@inheritDoc}
   */
  public Mirror(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public void update(Bitmap bmp) {
    this.bmp=bmp;
    invalidate();
```

```
  }

  /**
   * {@inheritDoc}
   */
  @Override
  protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    if (bmp != null) {
      getDrawingRect(rect);

      calcCenter(rect.width(), rect.height(), bmp.getWidth(),
                 bmp.getHeight(), rect);
      canvas.drawBitmap(bmp, null, rect, null);
    }
  }

  // based upon http://stackoverflow.com/a/14679729/115145

  static void calcCenter(int vw, int vh, int iw, int ih, Rect out) {
    double scale=
        Math.min((double)vw / (double)iw, (double)vh / (double)ih);

    int h=(int)(scale * ih);
    int w=(int)(scale * iw);
    int x=((vw - w) >> 1);
    int y=((vh - h) >> 1);

    out.set(x, y, x + w, y + h);
  }
}
```

The bulk of the "business logic" lies in `onDraw()`, plus a helper `calcCenter()` static method.

`onDraw()` is called on a `View` when it is time for that widget to actually draw its visual representation onto the supplied `Canvas`. Different widgets will use different drawing primitive methods offered by `Canvas`, to draw lines and text and whatnot. In our case, we:

- Calculate a `Rect` object with our size and position, using `getDrawingRect()`
- Get the `Bitmap` object from the `MirroringFrameLayout`, via a call to `getLastBitmap()` (which simply returns the `Bitmap` that the `MirroringFrameLayout` is using)

- Call `calcCenter` to adjust our `Rect` to take into account the fact that our size may be different than the size of the actual `Bitmap`
- Call `drawBitmap()` on our `Canvas`, to render the `Bitmap` into the location specified by the `Rect`, where `drawBitmap()` will automatically down-sample or up-sample the image as needed to fill the necessary space

Usage and Results

Normally, you would use the `Mirror` in a layout for a `Presentation` and the `MirroringFrameLayout` in an activity that controls the `Presentation`. However, it is possible to use both in the same layout file, for light testing. However, please do not put the `Mirror` inside of the `MirroringFrameLayout`, as this is likely to cause a rupture in the space-time continuum, and you really do not want to be responsible for that.

So, in the `SimpleMirrorActivity` from the `demo/` sub-project, we use a layout that has both `Mirror` and `MirroringFrameLayout`, with the latter set to mirror a `WebView`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context=".SimpleMirrorActivity">

  <FrameLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1">

    <com.commonsware.cwac.layouts.MirroringFrameLayout
      android:id="@+id/source"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_gravity="center">

      <EditText
        android:id="@+id/editor"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="left|top"
        android:inputType="textMultiLine"/>
    </com.commonsware.cwac.layouts.MirroringFrameLayout>
  </FrameLayout>
```

```
  <View
    android:layout_width="match_parent"
    android:layout_height="4dip"
    android:background="#FF000000"/>

  <com.commonsware.cwac.layouts.Mirror
    android:id="@+id/target"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="2"/>

</LinearLayout>
```

In this case, we set the background of the `FrameLayout` holding our `MirroringFrameLayout` to green, to show how the `MirroringFrameLayout` size is changed to maintain our aspect ratio.

(or, perhaps we just like green)

Besides configuring the to-be-mirrored widgets, all you need to do is call `setMirror()` on the `MirroringFrameLayout` to enable the mirroring logic:

```
package com.commonsware.cwac.layouts.demo;

import android.app.Activity;
import android.os.Bundle;
import com.commonsware.cwac.layouts.Mirror;
import com.commonsware.cwac.layouts.MirroringFrameLayout;

public class SimpleMirrorActivity extends Activity {
  MirroringFrameLayout source=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.simple_mirror);

    source=(MirroringFrameLayout)findViewById(R.id.source);
    Mirror target=(Mirror)findViewById(R.id.target);

    source.setMirror(target);
  }
}
```

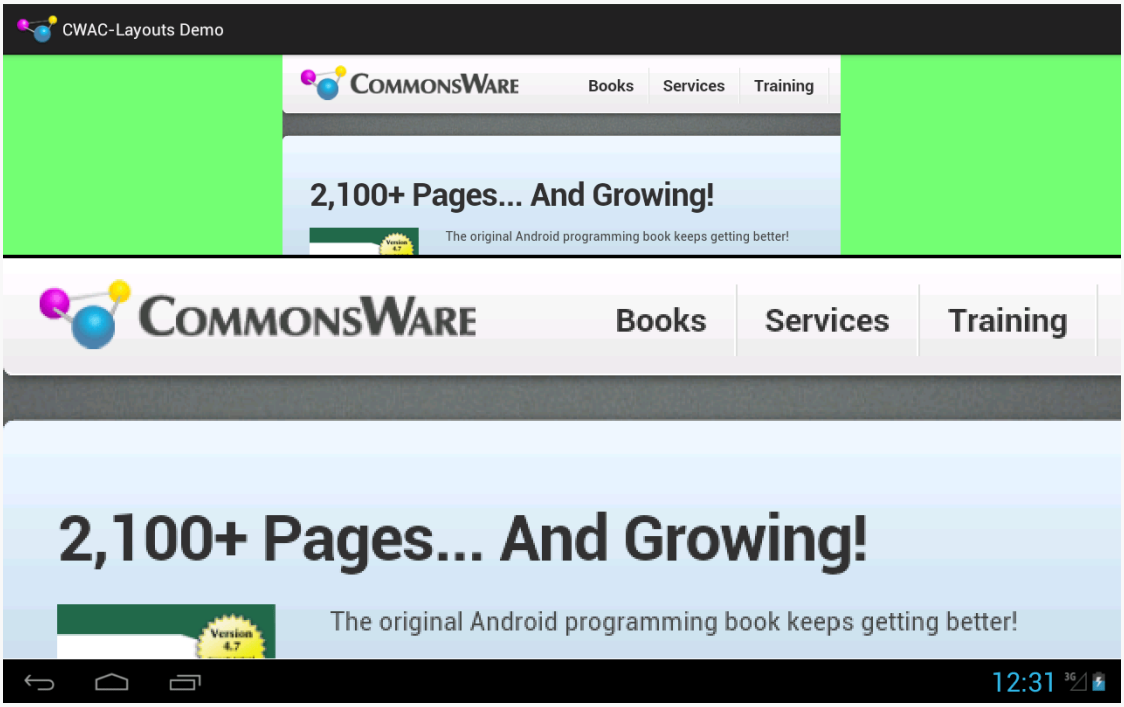


Figure 622: MirroringFrameLayout Above Its Mirror

While the bottom portion is just the `Mirror` and therefore is non-interactive, the top is the real `WebView`, which can be scrolled, with the resulting changes reflected in the `Mirror` in real-time:

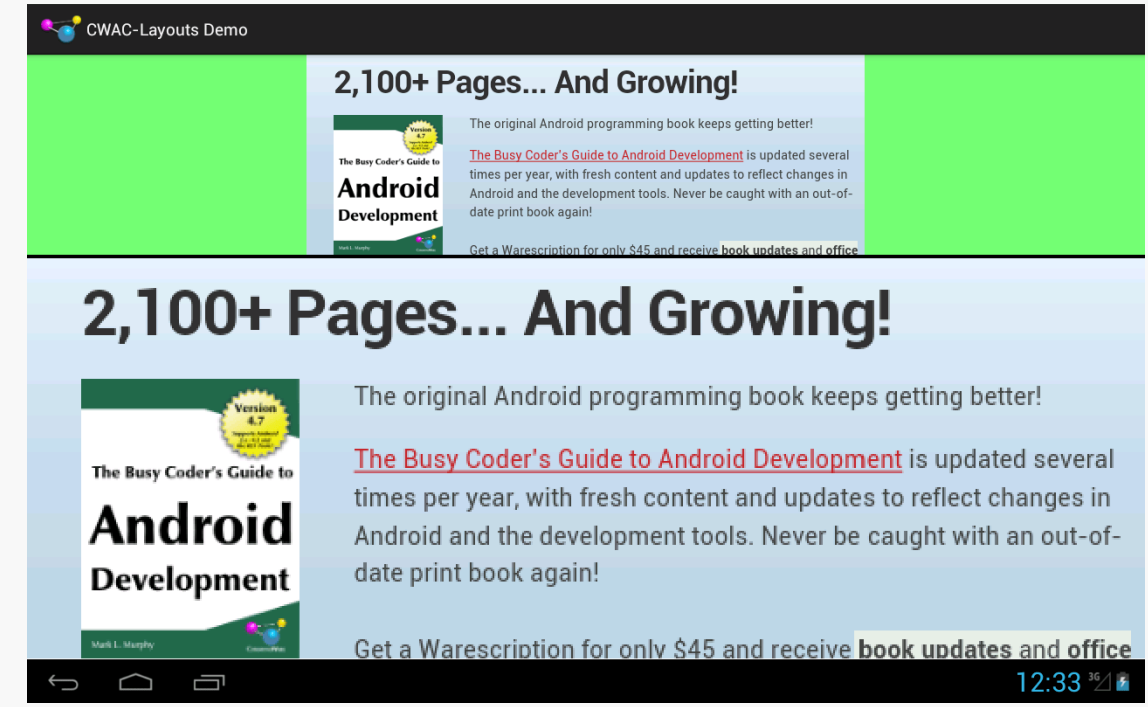


Figure 623: MirroringFrameLayout and Mirror, Showing Scrolled Contents

Limitations

`MirroringFrameLayout` only works for materials drawn in the Java layer, that therefore can be drawn to the `Bitmap`-backed `Canvas`. Content *not* drawn in the Java layer will not work with `MirroringFrameLayout`, notably anything involving a `SurfaceView`. This not only includes your own `SurfaceView` widgets, but anything else that depends upon `SurfaceView`, such as `VideoView` or the Maps V2 `MapView` and `MapFragment`.

Also, the re-sampling done by `Mirror` is not especially sophisticated and will cause jagged effects, particularly when up-sampling. Ideally, the `MirroredFrameLayout` will be the same size or larger than the `Mirror`. This may not always be possible, particularly with a `Mirror` shown on a 1080p external display, but the closer you can get will improve the output.

Advanced Preferences

We saw `SharedPreferences` and `PreferenceFragment` earlier in the book. However, we can have more elaborate preference collection options if we wish, such as a full master-detail implementation like the Settings app sports. There are also many other common attributes on the preference XML elements that we might consider taking advantage of, such as allowing us to automatically enable and disable preferences based upon whether some other preference is checked or unchecked.

In this chapter, we will explore some of these additional capabilities in the world of Android preferences.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on `SharedPreferences`.

Introducing PreferenceActivity

If you have a fairly simple set of preferences to collect from the user, using a single `PreferenceFragment` should be sufficient.

On the far other end of the spectrum, Android's Settings app collects a massive amount of preference values from the user. These are spread across a series of groups of preferences, known as preference headers.

While your app may not need to collect as many preferences as does the Settings app, you may need more than what could be collected easily in a single `PreferenceFragment`. In that case, you can consider adopting the same structure of

headers-and-fragments that the Settings app uses, by means of a `PreferenceActivity`.

To see this in action, take a look at the `Prefs/FragmentsBC` sample project. It is very similar to the original `SharedPreferences` demo app from before. However, this one arranges to collect a fifth preference value, in a separate `PreferenceFragment`, and uses `PreferenceActivity` to allow access to both `PreferenceFragment` UI structures.

Defining Your Preference Headers

In the master-detail approach offered by `PreferenceActivity`, the "master" list is a collection of preference headers. Typically, you would define these in another XML resource. In the sample project, that is found in `res/xml/preference_headers.xml`:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">

  <header
    android:fragment="com.commonsware.android.preffragsbc.EditPreferences$First"
    android:summary="@string/header1summary"
    android:title="@string/header1title">
  </header>
  <header
    android:fragment="com.commonsware.android.preffragsbc.EditPreferences$Second"
    android:summary="@string/header2summary"
    android:title="@string/header2title">
  </header>

</preference-headers>
```

(from Prefs/FragmentsBC/app/src/main/res/xml/preference_headers.xml)

Here, your root element is `<preference-headers>`, containing a series of `<header>` elements. Each `<header>` contains at least three attributes:

1. `android:fragment`, which identifies the Java class implementing the `PreferenceFragment` to use for this header, as is described in the next section
2. `android:title`, which is a few words identifying this header to the user

Once again, you may wish to also include `android:summary`, which is a short sentence explaining what the user will find inside of this header.

You can, if you wish, include one or more `<extra>` child elements inside the `<header>` element. These values will be put into the "arguments" `Bundle` that the associated `PreferenceFragment` can retrieve via `getArguments()`.

Creating Your PreferenceActivity

`EditPreferences` — which in the original sample app was a regular `Activity` — is now a `PreferenceActivity`. It contains little more than the two fragments referenced in the above preference header XML:

```
package com.commonsware.android.preffragsbc;

import android.os.Bundle;
import android.preference.PreferenceActivity;
import android.preference.PreferenceFragment;
import java.util.List;

public class EditPreferences extends PreferenceActivity {
  @Override
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
    if (First.class.getName().equals(fragmentName)
        || Second.class.getName().equals(fragmentName)) {
      return(true);
    }

    return(false);
  }

  public static class First extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      addPreferencesFromResource(R.xml.preferences);
    }
  }

  public static class Second extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
      addPreferencesFromResource(R.xml.preferences2);
    }
  }
}
```

(from Prefs/FragmentsBC/app/src/main/java/com/commonsware/android/preffragsbc/EditPreferences.java)

`onBuildHeaders()` is where we supply the preference headers, via a call to `loadHeadersFromResource()`.

We also need to have an `isValidFragment()` method, that will return `true` if the supplied fragment name is one we should be showing in this `PreferenceActivity`, `false` otherwise. This will only be called on Android 4.4+. However, we need to set up the project build target (e.g., `compileSdkVersion` in Android Studio) to API Level 19 or higher. Failing to have this method will cause your app to crash on Android 4.4+ devices, when the user tries to bring up one of your `PreferenceFragments`.

Each `PreferenceFragment` is then responsible for calling `addPreferencesFromResource()` to populate its contents. In this case, we now have two such resources: `res/xml/preferences.xml` (the original, used by `First`) and `res/xml/preferences2.xml` (used by `Second`).

The Results

On a wide enough screen — like that of a Nexus 9 in landscape — we get a master-detail presentation:

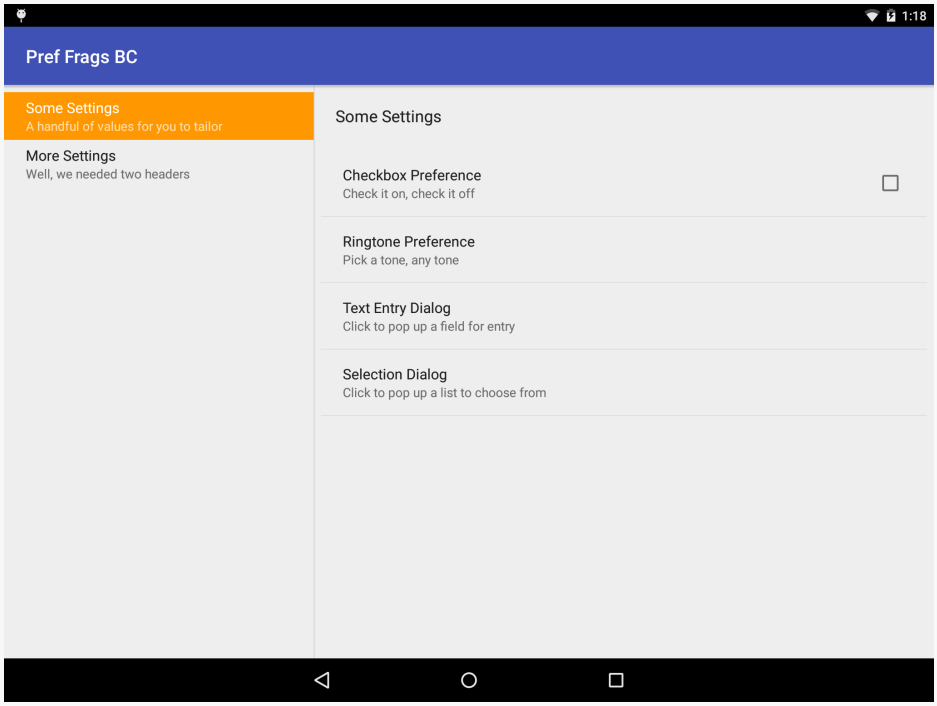


Figure 624: PreferenceActivity UI, on a Landscape Nexus 9

Here, we see the first preference fragment already pre-selected, showing its settings. Tapping on the second header will show the other preferences.

On a smaller screen, the master-detail approach means that we see a list of headers first:

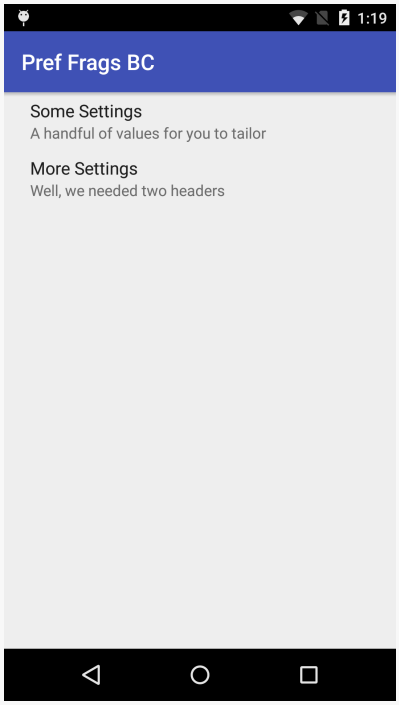


Figure 625: PreferenceActivity UI, on a Portrait Nexus 5

Tapping the headers give us access to the individual fragments.

Intents for Headers or Preferences

If you have the need to collect some preferences that are beyond what the standard preferences can handle, you have some choices.

One is to create a custom `Preference`. Extending `DialogPreference` to create your own `Preference` implementation is not especially hard. However, it does constrain you to something that can fit in a dialog.

Another option is to specify an `<intent>` element as a child of a `<header>` element. When the user taps on this header, your specified `Intent` is used with `startActivity()`, giving you a gateway to your own activity for collecting things that are beyond what the preference UI can handle. For example, you could have the following `<header>`:

```
<header android:icon="@drawable/something"
        android:title="Fancy Stuff"
        android:summary="Click here to transcend your
```

```
plane of existence">
  <intent android:action="com.commonsware.android.MY_CUSTOM_ACTION" />
</header>
```

Then, so long as you have an activity with an `<intent-filter>` specifying your desired action (`com.commonsware.android.MY_CUSTOM_ACTION`), that activity will get control when the user taps on the associated header.

Conditional Headers

The two-tier, headers-and-preferences approach is fine and helps to organize large rosters of preferences. However, it does tend to steer developers in the direction of displaying headers *all of the time*. For many apps, that is rather pointless, because there are too few preferences to collect to warrant having more than one header.

One alternative approach is to use the headers on larger devices, but skip them on smaller devices. That way, the user does not have to tap past a single-item `ListFragment` just to get to the actual preferences to adjust.

This is a wee bit tricky to implement. However, you have two options for how to accomplish it.

(The author would like to thank Richard Le Mesurier, whose question on this topic spurred the development of this section and its samples)

Option #1: Do Not Define the Headers

The basic plan in the first approach is to have smarts in `onBuildHeaders()` to handle this. `onBuildHeaders()` is the callback that Android invokes on our `PreferenceActivity` to let us define the headers to use in the master-detail pattern. If we want to have headers, we would supply them here; if we want to skip the headers, we would instead fall back to the classic (and, admittedly, deprecated) `addPreferencesFromResource()` method to load up some preference XML.

There is an `isMultiPane()` method on `PreferenceActivity`, starting with API Level 11, that will tell you if the activity will render with two fragments (master+detail) or not. In principle, this would be ideal to use. Unfortunately, it does not seem to be designed to be called from `onBuildHeaders()`. Similarly, `addPreferencesFromResource()` does not seem to be callable from `onBuildHeaders()`. Both are due to timing: `onBuildHeaders()` is called in the middle of the `PreferenceActivity` `onCreate()` processing.

So, we have to do some fancy footwork.

By examining the source code to `PreferenceActivity`, you will see that the logic that drives the single-pane vs. dual-pane UI decision boils down to:

```
onIsHidingHeaders() || !onIsMultiPane()
```

If that expression returns `true`, we are in single-pane mode; otherwise, we are in dual-pane mode. `onIsHidingHeaders()` will normally return `false`, while `onIsMultiPane()` will return either `true` or `false` based upon screen size.

So, we can leverage this information in a `PreferenceActivity` to conditionally load our headers, as seen in the `EditPreferences` class in the `Prefs/SingleHeader` sample project:

```
package com.commonsware.android.pref1header;

import android.os.Bundle;
import android.preference.PreferenceActivity;
import java.util.List;

public class EditPreferences extends PreferenceActivity {
  private boolean needResource=false;

  @SuppressWarnings("deprecation")
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (needResource) {
      addPreferencesFromResource(R.xml.preferences);
    }
  }

  @Override
  public void onBuildHeaders(List<Header> target) {
    if (onIsHidingHeaders() || !onIsMultiPane()) {
      needResource=true;
    }
    else {
      loadHeadersFromResource(R.xml.preference_headers, target);
    }
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
```

```
    return(StockPreferenceFragment.class.getName().equals(fragmentName));
  }
}
```

(from Prefs/SingleHeader/app/src/main/java/com/commonsware/android/prefiheader/EditPreferences.java)

Here, if we are in dual-pane mode, `onBuildHeaders()` populates the headers as normal. If, though, we are in single-pane mode, we skip that step and make note that we need to do some more work in `onCreate()`.

Then, in `onCreate()`, if we did not load our headers we use the classic `addPreferencesFromResource()` method.

The net result is that on Android 3.0+ tablets, we get the dual-pane, master-detail look with our one header, but on smaller devices (regardless of version), we roll straight to the preferences themselves.

Note that this sample application uses a single `PreferenceFragment` implementation, named `StockPreferenceFragment`:

```
package com.commonsware.android.pref1header;

import android.os.Bundle;
import android.preference.PreferenceFragment;

public class StockPreferenceFragment extends PreferenceFragment {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    int res=
        getActivity().getResources()
                     .getIdentifier(getArguments().getString("resource"),
                                    "xml",
                                    getActivity().getPackageName());

    addPreferencesFromResource(res);
  }
}
```

(from Prefs/SingleHeader/app/src/main/java/com/commonsware/android/prefiheader/StockPreferenceFragment.java)

`StockPreferenceFragment` does what it is supposed to: call `addPreferencesFromResource()` in `onCreate()` with the resource ID of the

preferences to load. However, rather than hard-coding a resource ID, as we normally would, we look it up at runtime.

The `<extra>` elements in our preference header XML supply the name of the preference XML to be loaded:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">

  <header
    android:fragment="com.commonsware.android.pref1header.StockPreferenceFragment"
    android:summary="@string/header1summary"
    android:title="@string/header1title">
    <extra
      android:name="resource"
      android:value="preferences"/>
  </header>

</preference-headers>
```

(from Prefs/SingleHeader/app/src/main/res/xml/preference_headers.xml)

We get that name via the arguments `Bundle` (`getArguments().getString("resource")`).

To look up a resource ID at runtime, we can use the `Resources` object, available from our activity via a call to `getResources()`. `Resources` has a method, `getIdentifier()`, that will return a resource ID given three pieces of information:

1. The base name of the resource (in our case, the value retrieved from the `<extra>` element)
2. The type of the resource (e.g., `"xml"`)
3. The package holding the resource (in our case, our own package, retrieved from our activity via `getPackageName()`)

Note that `getIdentifier()` uses reflection to find this value, and so there is some overhead in the process. Do not use `getIdentifier()` in a long loop – cache the value instead.

The net is that `StockPreferenceFragment` loads the preference XML described in the `<extra>` element, so we do not need to create separate `PreferenceFragment` implementations per preference header.

Option #2: Go Directly to the Fragment

The advantage of the above approach is that it works with Android's own logic of whether to display the master-detail fragments or just one at a time. However, that logic — the fact that `onIsHidingHeaders() || !onIsMultiPane()` determines the look of the activity — is not documented, and therefore may change in future Android releases.

Another option is to launch your `PreferenceActivity` in such a way that tells Android to skip showing the headers. This approach is better documented and therefore perhaps more stable. This can also be used in cases where you do want headers sometimes, but at other times you want to route the user to a specific `PreferenceFragment`. The downside is that this technique only works on API Level 11+.

To see how this works, take a look at the `Prefs/SingleHeader2` sample project.

Our `EditPreferences` class is the same implementation as in the original sample for this chapter, except that we only load up the single XML resource's worth of preferences:

```
package com.commonsware.android.pref1header;

import android.preference.PreferenceActivity;
import java.util.List;

public class EditPreferences extends PreferenceActivity {
  @Override
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
    return(StockPreferenceFragment.class.getName().equals(fragmentName));
  }
}
```

(from Prefs/SingleHeader2/app/src/main/java/com/commonsware/android/prefiheader/EditPreferences.java)

However, there is a change in our main activity (`FragmentsDemo`). Before, when the user chose the "Settings" action bar overflow item, we would just call `startActivity()` to bring up `EditPreferences`. Now, we delegate that work to an

`editPrefs()` method on `FragmentsDemo`, which will have the smarts to control *how* we bring up the `EditPreferences` activity:

```
private void editPrefs() {
  Intent i=new Intent(this, EditPreferences.class);

  i.putExtra(PreferenceActivity.EXTRA_SHOW_FRAGMENT,
             StockPreferenceFragment.class.getName());

  Bundle b=new Bundle();

  b.putString("resource", "preferences");

  i.putExtra(PreferenceActivity.EXTRA_SHOW_FRAGMENT_ARGUMENTS, b);

  startActivity(i);
}
```

(from Prefs/SingleHeader2/app/src/main/java/com/commonsware/android/prefheader/FragmentsDemo.java)

Here, we will add two extras to our `Intent`:

- `EXTRA_SHOW_FRAGMENT`, set to the fully-qualified class name of the `PreferenceFragment` to be displayed, here obtained by calling `getName()` on the `Class` object for `StockPreferenceFragment`
- `EXTRA_SHOW_FRAGMENT_ARGUMENTS`, set to a `Bundle` containing the same values that would ordinarily be loaded from the `<extra>` elements in the preference header XML resource (in our case, the name of the preference XML resource to load)

Those extras will be automatically handled by `PreferenceActivity` (on API Level 11+) and will have the effect of directly taking the user to our one-and-only fragment, bypassing the headers.

Dependent Preferences

In the Settings app, or in other apps that appear to be using `PreferenceFragment`-based UIs, you may have noticed that there are times when preferences are disabled. They become enabled when you check a `CheckBoxPreference` or toggle on a `SwitchPreference`.

That is handled via the `android:dependency` attribute on the to-be-disabled preferences. The value of `android:dependency` is the key of a `TwoStatePreference`

subclass, such as a `CheckBoxPreference` or a `SwitchPreference`. The enabled/disabled state of the preference with the `android:dependency` attribute depends on the checked state of the named dependency.

For example, the `Prefs/Dependency` sample project is a clone of the original `SharedPreferences` demo app with one slight change: all the preferences other than `checkbox` are now dependent upon `checkbox`:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

  <CheckBoxPreference
    android:key="checkbox"
    android:summary="@string/pref1summary"
    android:title="@string/pref1title"/>

  <RingtonePreference
    android:dependency="checkbox"
    android:key="ringtone"
    android:showDefault="true"
    android:showSilent="true"
    android:summary="@string/pref2summary"
    android:title="@string/pref2title"/>

  <EditTextPreference
    android:dependency="checkbox"
    android:dialogTitle="@string/dialogtitle"
    android:key="text"
    android:summary="@string/pref3summary"
    android:title="@string/pref3title"/>

  <ListPreference
    android:dependency="checkbox"
    android:dialogTitle="@string/listdialogtitle"
    android:entries="@array/cities"
    android:entryValues="@array/airport_codes"
    android:key="list"
    android:summary="@string/pref4summary"
    android:title="@string/pref4title"/>

</PreferenceScreen>
```

(from Prefs/Dependency/app/src/main/res/xml/preferences.xml)

When you run the project, the dependent preferences are disabled while the checkbox is unchecked:

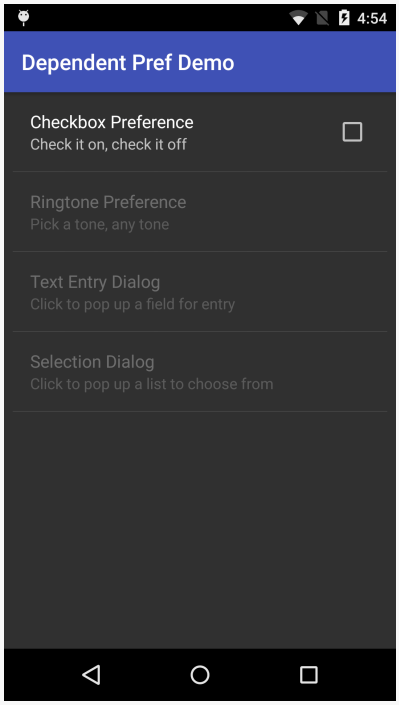


Figure 626: Dependent Preferences, Disabled

...but become enabled once the user checks the checkbox:

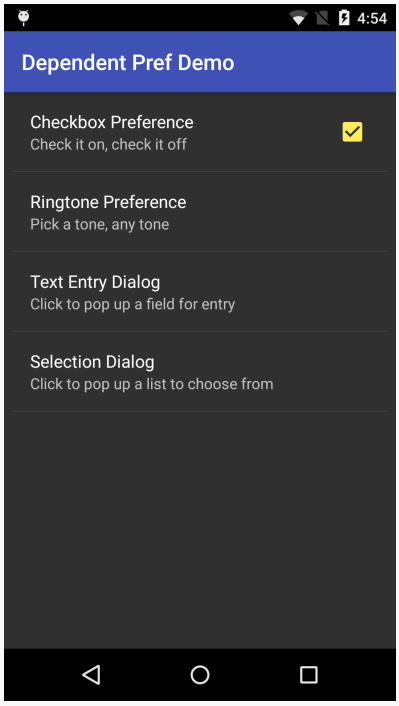


Figure 627: Dependent Preferences, Enabled

Nested Screens

Perhaps you have more preferences than you want to collect on a single screen, but you do not feel that a master-detail presentation is the right structure. Or, perhaps you have *lots* of preferences to collect, and even collecting preferences into groups by header is insufficient.

Another possibility is to nest preference screens. One screen holds another. On the outer preference screen, the user has a "preference" entry that simply displays the nested screen, as opposed to directly collecting any preferences.

A `<PreferenceScreen>` element in your preference XML can hold another `<PreferenceScreen>` element. That inner `<PreferenceScreen>` can come in one of two forms:

1. Inside the inner `<PreferenceScreen>` you have more preference XML elements. This means there is only one `PreferenceFragment` for the whole structure (outer `<PreferenceScreen>`, including the inner

<PreferenceScreen>). However, visually, the user will "drill down" from the outer screen into the inner one by tapping on an entry.

2. The inner <PreferenceScreen> has an android:fragment attribute, just like a preference header might. This points to a Fragment — typically a PreferenceFragment — that will be responsible for the "inner" content. This is a bit more complex to set up, as it requires a couple of fragments. However, it gives you greater flexibility. Plus, it is fairly easy to then switch from using preference headers and the master-detail approach to using nested preference screens, or back again, as you are simply reusing the same PreferenceFragment implementations in either case.

The Prefs/NestedScreens sample project takes the master-detail approach shown earlier in this chapter and switches it to having a top-level screen and a nested screen. This is accomplished by adding a <PreferenceScreen> element to res/xml/preferences.xml, pointing to our Second PreferenceFragment:

```
<PreferenceScreen
  android:fragment="com.commonsware.android.preffragsbc.EditPreferences$Second"
  android:key="unused"
  android:title="@string/nested_title"/>
```

(from Prefs/NestedScreens/app/src/main/res/xml/preferences.xml)

Here, the android:title (and optional android:summary) will be shown on the *outer* screen, as an entry that the user can tap on to get to this inner screen. While in this sample, we are not using android:key, in principle you could use this to get at the PreferenceScreen itself to manipulate it at runtime (e.g., disable it).

For this style of <PreferenceScreen> to work, the preference XML must be used by a PreferenceFragment in a PreferenceActivity — you cannot use it with a regular Activity. However, just because you use PreferenceActivity does not mean that you have to opt into the master-detail structure. We can use the same onCreate(), show-the-PreferenceFragment approach that we use with a regular Activity.

However, there is one big catch: when the user taps on the entry that will launch the inner screen, the Android framework will start *another instance* of our PreferenceActivity. It will give us the same EXTRA_SHOW_FRAGMENT value as we saw earlier in this chapter. However, PreferenceActivity will automatically show that fragment; we do not need to show it ourselves.

But, this means that our onCreate() needs to distinguish between the "show the outer screen ourselves" case and the "show the inner screen automatically" case, which we can do by seeing if EXTRA_SHOW_FRAGMENT exists:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (getIntent().getStringExtra(PreferenceActivity.EXTRA_SHOW_FRAGMENT)==null) {
    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager().beginTransaction()
          .add(android.R.id.content,
              new First()).commit();
    }
  }
}
```

(from Prefs/NestedScreens/app/src/main/java/com/commonsware/android/preffragsbc/EditPreferences.java)

The result is that we see the outer screen first, containing our entry for the inner screen:

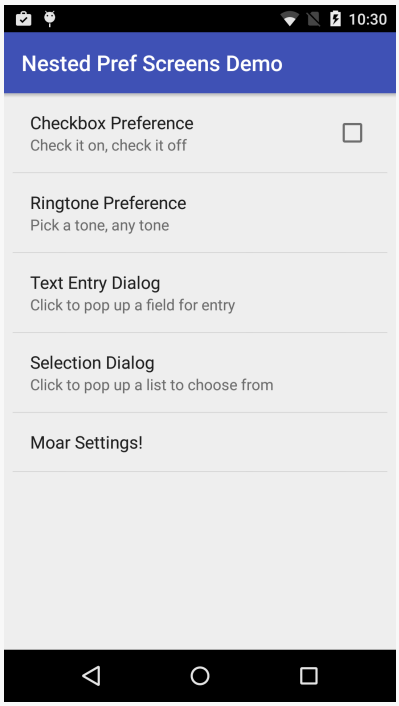


Figure 628: Nested Preferences, Outer Screen

Tapping on that entry brings up the inner, nested, screen:

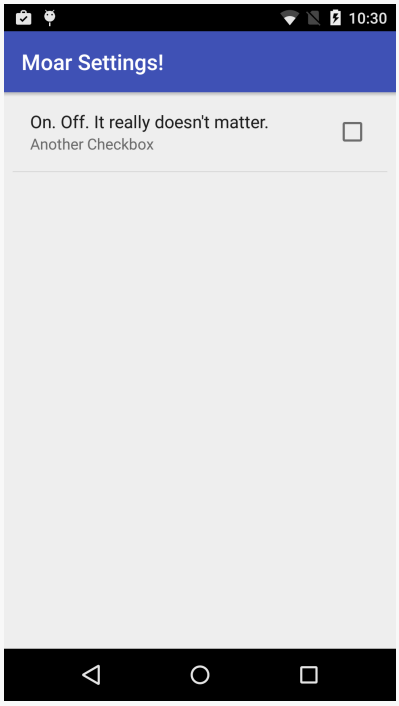


Figure 629: Nested Preferences, Inner Screen

Listening to Preference Changes

Sometimes, you may need to take steps when the user interacts with a preference in your `PreferenceFragment`-based UI.

A common scenario for this comes with the summary. In some cases, is it handy to have the summary reflect the current value of the preference. While some preferences naturally show their value inline (e.g., a `CheckBoxPreference`), those that extend from `DialogPreference` only show their value when the user taps on the preference to display the dialog. Putting something in the summary that reflects the value can save the user a click.

However, by default, the summary is static, populated by the `android:summary` attribute in your preference XML. If you want it to reflect the current preference value, you not only need to be able to set the summary in Java, but to be able to respond when the user changes the value, so you can update the summary again.

The `Prefs/CustomSubtitle` sample project demonstrates how this works. This is yet another clone of the original `SharedPreferences` demo app. This time, the

preference XML is unchanged from the original. However, we have a slightly more elaborate `PreferenceFragment` implementation:

```
public static class Prefs extends PreferenceFragment
    implements Preference.OnPreferenceChangeListener {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences);

    Preference pref=findPreference("text");

    updateSummary(pref,
        pref.getSharedPreferences().getString(pref.getKey(), null));
    pref.setOnPreferenceChangeListener(this);
  }

  @Override
  public boolean onPreferenceChange(Preference pref, Object newValue) {
    updateSummary(pref, newValue.toString());

    return(true);
  }

  private void updateSummary(Preference pref, String value) {
    if (value==null || value.length()==0) {
      pref.setSummary(R.string.msg_missing_text);
    }
    else {
      pref.setSummary(value);
    }
  }
}
```

(from Prefs/CustomSubtitle/app/src/main/java/com/commonsware/android/preffrag/EditPreferences.java)

In `onCreate()`, after `addPreferencesFromResource()`, we call `findPreference()` to retrieve the `Preference` object that manages the snippet of UI for a particular preference. The flow here mimics that of `setContentView()` and `findViewById()`: first you inflate the resource, then you find the Java object corresponding to some XML element out of that resource. `findPreference()` takes the key of the preference that you are looking for; in this case, we are looking for the `EditTextPreference`, whose key is `text`.

We then call a private `updateSummary()` method, which takes the `Preference` and the current value of that preference and updates the summary. To get the current value, `onCreate()` can ask the `Preference` for its backing `SharedPreferences` (via `getSharedPreferences()`), then retrieve the value using standard getters (e.g., `getString()`). `updateSummary()` then shows the string representation of the current value, or a canned message if there does not appear to be a current value.

We also register the fragment itself as being the `OnPreferenceChangeListener`, and register the fragment with the preference via `setOnPreferenceChangeListener()`. This means that when the user manipulates this preference, we will be called with `onPreferenceChange()`. This is done *before the `SharedPreferences` are updated*. Our options are either to return `true` and have the normal persistence process continue, or return `false` and manage persistence ourselves (e.g., perform some conversion on the raw value before storing it). In our case, we are just using this to call `updateSummary()` again.

If you install the app and run it, you will not have an existing value for the preference, and so the summary shows a stock message:

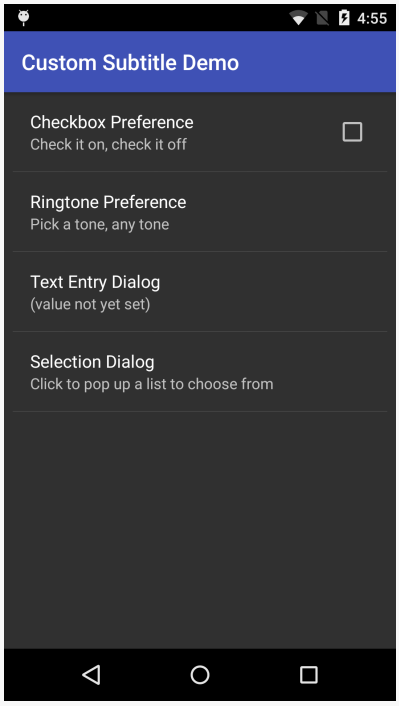


Figure 630: Custom Subtitle Demo, Before Editing Text

After you tap on the `EditTextPreference` and fill in some value in the dialog, the summary updates to show what you typed in:

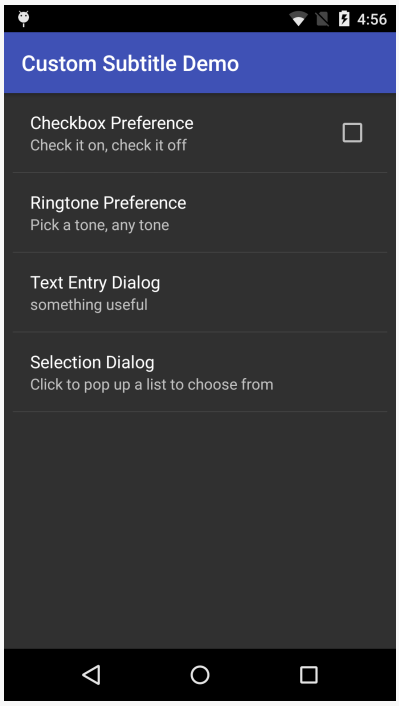


Figure 631: Custom Subtitle Demo, After Editing Text

Defaults, and Defaults

When you use `SharedPreferences` to retrieve a value, you can usually provide a default value along with the key for the value that you want. If there is no preference value for that key, you get the default that you supplied.

A preference in preference XML also has an `android:defaultValue` attribute. This is, roughly speaking, the preference UI counterpart to that second parameter to the `SharedPreferences` getters. If the user interacts with the preference, the `android:defaultValue` value will be presented to the user if there was no preference value stored for that key in the underlying `SharedPreferences`.

To synchronize these, you can call `setDefaultValues()` on the `PreferenceManager` class. Given the resource ID of some preference XML, `PreferenceManager` will find all `android:defaultValue` attributes and then persist those default values to the `SharedPreferences` under their respective keys.

Listening to Preference Value Changes

Sometimes, you will have components that need to know when preference values are changed elsewhere in your app. For example, you may have a `Service` that is using information from `SharedPreferences`, and the `Service` may need to know when those values change.

One approach, used in all the sample apps, is simply to re-read the preference values as needed, rather than caching them in data members or something. After the first time `SharedPreferences` are accessed, the `SharedPreferences` themselves are held in heap space, and so accessing them can be fairly cheap. So, the sample apps' launcher activities just re-read the preference values in `onResume()` and update the UI that way.

If, however, that is inappropriate, inconvenient, or otherwise not what you want to do, you can call `registerOnSharedPreferenceChangeListener()` on a `SharedPreferences` object, supplying an instance of an implementation of the `OnSharedPreferenceChangeListener` interface. That object will be called with `onSharedPreferenceChanged()` every time a preference value changes. You are given the key to the changed value, so you can implement a filter to only pay attention to keys that matter to you. When one of those keys is reported to have changed, you can ask the `SharedPreferences` for the new value.

Dynamic ListPreference Contents

Many times, the items that the user can choose from in your `ListPreference` or `MultiSelectListPreference` are fixed, allowing you to populate them from `<string-array>` resources. However, sometimes, the items (display names and corresponding values) are dynamic, based upon information held elsewhere: database, server, or something at a system level. For those, we need to be able to define the preference in XML, but configure its contents in Java code.

For example, the `Introspection/SAWMonitor` sample project is a monitor for new and upgraded apps that ask for the `SYSTEM_ALERT_WINDOW` permission. Such apps have the right to draw over top of other apps, for anything from Facebook "chatheads" to tapjacking attacks.

However, some apps may request this permission that you are perfectly fine with having it. By default, SAWMonitor will point out this permission on each subsequent update, which can get tiresome after a while. Hence, SAWMonitor

allows you to add apps to a "whitelist"; those apps will be ignored, even if they request SYSTEM_ALERT_WINDOW.

To that end, we have a settings.xml resource describing some preferences to collect from the user:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
  <SwitchPreference
    android:key="enabled"
    android:title="@string/msg_enable"
    android:defaultValue="true"/>
  <MultiSelectListPreference
    android:key="whitelist"
    android:title="@string/msg_whitelist" />
</PreferenceScreen>
```

(from Introspection/SAWMonitor/app/src/main/res/xml/settings.xml)

Here we have two preferences: a SwitchPreference for whether we should be monitoring for SYSTEM_ALERT_WINDOW at all, and a MultiSelectListPreference to allow the user to control the whitelist.

In onCreate() of our SettingsFragment, we load up those preferences into the UI via addPreferencesFromResource(), use findPreference() to retrieve both of the Preference objects, and use setOnPreferenceChangeListener() to be notified about changes to the enabled preference:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.settings);
    pm=getActivity().getPackageManager();

    enabled=(SwitchPreference)findPreference(MonitorApp.PREF_ENABLED);

    populateWhitelist((MultiSelectListPreference)findPreference("whitelist"));
  }
```

(from Introspection/SAWMonitor/app/src/main/java/com/commonsware/android/sawmonitor/SettingsFragment.java)

The populateWhitelist() call is where we fill in the details for the MultiSelectListPreference. In our case, the possible values are the apps presently installed that have requested the SYSTEM_ALERT_WINDOW permission. So, we use

PackageManager to find those, then use that information to populate the whitelist preference:

```
void populateWhitelist(MultiSelectListPreference whitelist) {
  List<ApplicationInfo> apps=pm.getInstalledApplications(0);

  Collections.sort(apps,
    new ApplicationInfo.DisplayNameComparator(pm));

  ArrayList<CharSequence> displayNames=
    new ArrayList<CharSequence>();
  ArrayList<String> packageNames=new ArrayList<String>();

  for (ApplicationInfo app : apps) {
    try {
      PackageInfo pkgInfo=
        pm.getPackageInfo(app.packageName,
          PackageManager.GET_PERMISSIONS);

      if (pkgInfo.requestedPermissions!=null) {
        for (String perm : pkgInfo.requestedPermissions) {
          if (SYSTEM_ALERT_WINDOW.equals(perm)) {
            displayNames.add(app.loadLabel(pm));
            packageNames.add(app.packageName);
            break;
          }
        }
      }
    }
    catch (PackageManager.NameNotFoundException e) {
      // should not happen, quietly ignore
    }
  }

  whitelist
    .setEntries(displayNames
      .toArray(new CharSequence[displayNames.size()]));
  whitelist
    .setEntryValues(packageNames
      .toArray(new String[packageNames.size()]));
}
```

(from Introspection/SAWMonitor/app/src/main/java/com/commonsware/android/sawmonitor/SettingsFragment.java)

Most of the code is determining which applications have that permission. However, MultiSelectListPreference complicates matters, by having two separate setter methods for its contents:

- `setEntries()` sets the display names, what the user will see in the multi-select dialog
- `setEntryValues()` sets the corresponding values, what will be stored in the `SharedPreferences` based upon the user's input

These each take arrays of `CharSequence` implementations, like `String`. Hence, we need two parallel arrays of values, rather than a single `ArrayList` of `Pair` objects or something.

With that in mind, `populateWhitelist()`:

- Gets the list of installed applications from the `PackageManager` (`pm` is a field initialized in `onCreate()`)
- Sorts those by display name, so our results will wind up in alphabetical order
- Creates an `ArrayList` for the display names and a separate one for the package names, which will serve as our entry values
- Iterates over the applications, gets the permissions requested by each app, and if any of those is `SYSTEM_ALERT_WINDOW`, add the display name (`loadLabel()`) and the package name to their respective lists
- Converts each of those `ArrayList` objects into a corresponding Java array, and passes them to their appropriate setters

The resulting `SettingsFragment` has the two preferences:

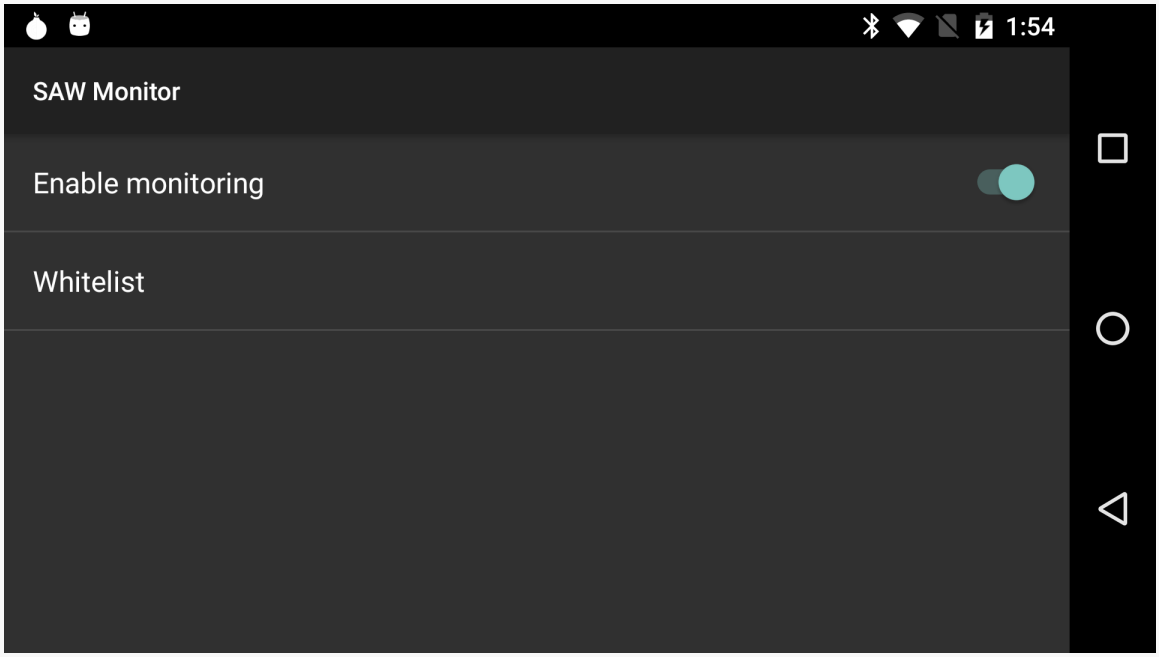


Figure 632: SAWMonitor SettingsFragment

Tapping on the "Whitelist" entry brings up the `MultiSelectListPreference`:

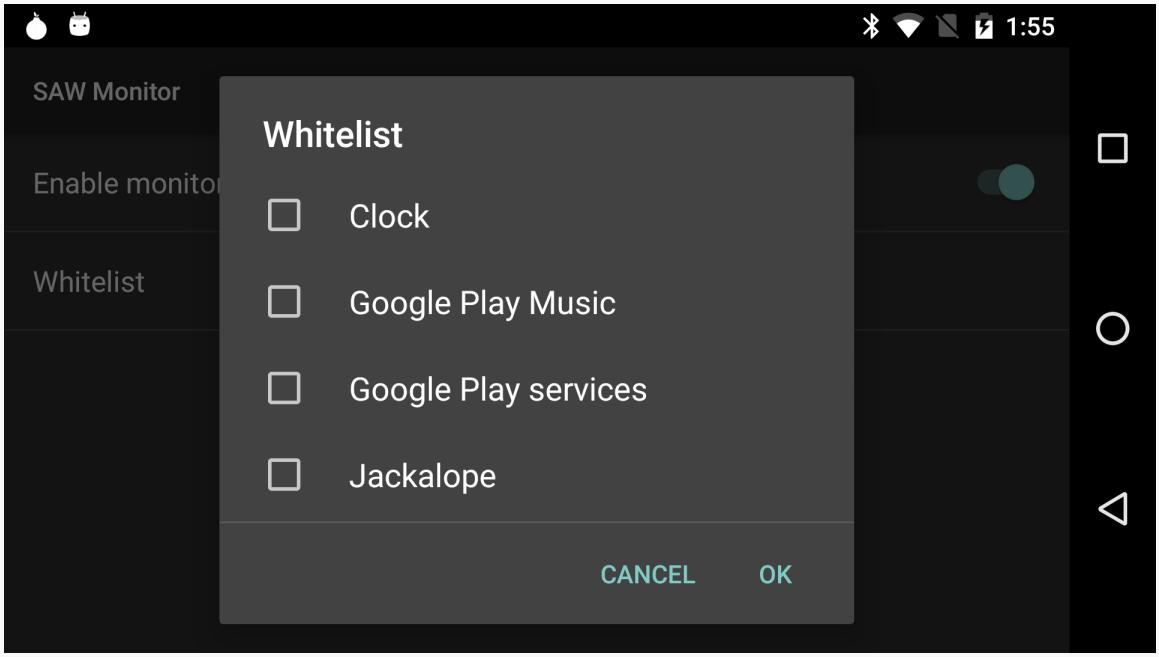


Figure 633: SAWMonitor Whitelist MultiSelectListPreference

If you run the app on your device or emulator, you will wind up with different possible entries in the `MultiSelectListPreference`, as the mix of apps requesting `SYSTEM_ALERT_WINDOW` will be different for different devices and users.

Dealing with External Changes to Preferences

What happens if you have a `PreferenceFragment` in the foreground, and the preference changes "behind the scenes" by some other component of your app?

For preferences with dialogs — `ListPreference`, `EditTextPreference`, etc. — the pattern seems to be "transaction by dialog". Whatever the preference value is at the time the dialog appears is what the user sees, and that does not change (and cannot readily *be* changed) if the preference changes while that dialog is on the screen.

However, for inline preferences — `CheckBoxPreference`, `SwitchPreference`, etc. — while the UI will not automatically update based on the external change, you can handle that yourself.

For example, `SAWMonitor` offers an optional notification shade tile using a `TileService` on Android 7.0+ devices. The tile allows the user to enable and disable the monitoring, just as the user can from the `SwitchPreference`. So... what happens if the `SettingsFragment` is on the screen, the user slides open the notification shade, and taps the tile? By default, the `SettingsFragment` would be oblivious to this, with the result of the `SwitchPreference` being out of sync.

But, we can fix this.

In `SettingsFragment`, in `onStart()`, we register for preference changes, plus call a `syncEnabledStates()` method. We unregister from preference changes in `onStop()`:

```
@Override
public void onStart() {
  super.onStart();

  prefs=PreferenceManager.getDefaultSharedPreferences(getActivity());
  prefs.registerOnSharedPreferenceChangeListener(this);
  syncEnabledStates();
}

@Override
public void onStop() {
```

```
    super.onStop();

    prefs.unregisterOnSharedPreferenceChangeListener(this);
  }
```

(from Introspection/SAWMonitor/app/src/main/java/com/commonsware/android/sawmonitor/SettingsFragment.java)

The `onSharedPreferenceChanged()` method on our `SettingsFragment` will be called when any of our preferences changes. If the `enabled` preference changes, we call `syncEnabledStates()`:

```
  @Override
  public void onSharedPreferenceChanged(SharedPreferences prefs,
                                        String s) {
    if (MonitorApp.PREF_ENABLED.equals(s)) {
      syncEnabledStates();
    }
  }
```

(from Introspection/SAWMonitor/app/src/main/java/com/commonsware/android/sawmonitor/SettingsFragment.java)

`syncEnabledStates()` simply updates the checked state of `enabled` based upon the now-current value in `SharedPreferences`:

```
  void syncEnabledStates() {
    enabled.setChecked(prefs.getBoolean(MonitorApp.PREF_ENABLED, false));
  }
```

(from Introspection/SAWMonitor/app/src/main/java/com/commonsware/android/sawmonitor/SettingsFragment.java)

Hence, this also handles the case where our `SettingsFragment` was displayed, the user navigated elsewhere, one of our preferences changes, and then the user returns to our running `SettingsFragment`. Normally, the `SettingsFragment` might miss that preference change, but with this implementation, the `SettingsFragment` will be kept in sync with the actual preference value.

Preferences in Device Settings App

On Android 7.0+, you can have the Settings app show a "gear" icon on your activity that collects preferences. When the user taps that gear, the Settings app will launch your designated activity:

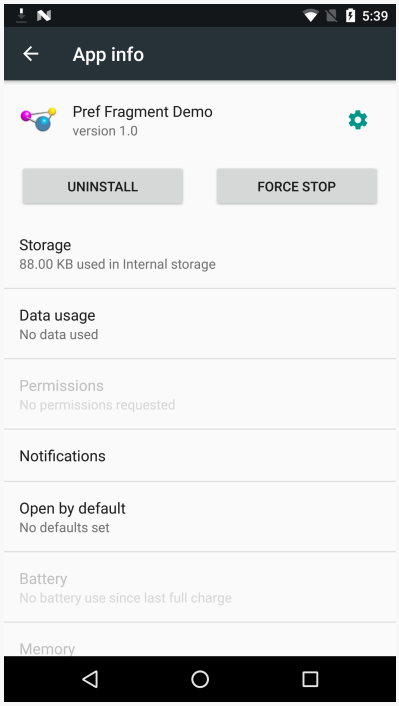


Figure 634: Settings Activity Gear Icon

To offer this, you need to add an `<intent-filter>` for your desired activity, with an `<action>` of `android.intent.action.APPLICATION_PREFERENCES`:

```
<activity
  android:name="EditPreferences"
  android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.APPLICATION_PREFERENCES" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

However, by default, there is a cost to this: *any app* can start your settings activity, whenever another app wants to. Your settings activity is exported once you add the `<intent-filter>`, and it needs to be exported for the Settings app to be able to start the activity.

However, it is fairly likely that this activity was not exported before you added this `<intent-filter>`. And while you may not mind it if the Settings app starts this activity, or if your own application code starts this activity, you may not want arbitrary other apps to start this activity. A general rule of thumb in modern

development is to keep your "attack surface" low. Having an activity be exported for little value is an unnecessary increase in your app's attack surface.

There is no officially-documented solution for this, though perhaps they will add one someday.

There are two candidate approaches. An unexpected one works: you can mark the activity as being not exported, via `android:exported="false"`. For some reason, the Settings app can still start up that activity, perhaps due to some system-level privilege. However, other apps will be unable to start the activity. This would result in an `<activity>` element like this:

```
<activity
  android:name="EditPreferences"
  android:label="@string/app_name"
  android:exported="false">
  <intent-filter>
    <action android:name="android.intent.action.APPLICATION_PREFERENCES" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Another approach that should work is to use `android:permission` to limit what other apps can start your activity, choosing a permission that the Settings app is sure to have but that most other apps will lack. `WRITE_SECURE_SETTINGS` is one candidate:

```
<activity
  android:name="EditPreferences"
  android:label="@string/app_name"
  android:permission="android.permission.WRITE_SECURE_SETTINGS">
  <intent-filter>
    <action android:name="android.intent.action.APPLICATION_PREFERENCES" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Now, the only other apps that can start your activity must hold the `WRITE_SECURE_SETTINGS` permission, which ordinary Android SDK apps cannot hold.

Custom Preference Storage

`SharedPreferences` are stored in an app's portion of internal storage as an XML file. This is fine in many cases. However, it is a problem for apps that need to ensure that persisted data is encrypted. While you can create a wrapper around `SharedPreferences` that encrypts the keys and values, that will not work well with things like the preference screen UI system. Basically, anything that does not know about the wrapper would try working with the actual `SharedPreference` data and be broken by the encryption.

This has always been disappointing, considering that `SharedPreferences` is an interface, and so setting up some sort of decorator approach should have been fairly easy to add.

In Android 8.0+, Google does not do that.

However, they *do* add in another mechanism: `PreferenceDataStore`.

You can create a `PreferenceDataStore` and associate it with a `PreferenceManager` via `setPreferenceDataStore()`. Then, all `SharedPreferences` loaded from that `PreferenceManager` will not use the normal XML-based persistence. Instead, the `PreferenceDataStore` will be used instead. That interface has getter and setter methods for all of the types supported by `SharedPreferences`, and it is the responsibility of some instance of `PreferenceDataStore` to handle the persistence as you see fit.

This solution is goofy, but it works, after a fashion, as is illustrated in the [Prefs/DataStore] sample project.

In that project, we have a `SillyDataStore` implementation of the `PreferenceDataStore` interface. It just stuffs all the data into a `HashMap`:

```
package com.commonsware.android.preffrag;

import android.preference.PreferenceDataStore;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

class SillyDataStore implements PreferenceDataStore {
  static private final SillyDataStore INSTANCE=new SillyDataStore();
  private Map<String, Object> cache=new HashMap<>();
```

```
static SillyDataStore get() {
  return(INSTANCE);
}

private SillyDataStore() {
  // just here to prevent accidental creation from outside
}

@Override
public void putString(String key, String value) {
  cache.put(key, value);
}

@Override
public void putStringSet(String key, Set<String> values) {
  cache.put(key, values);
}

@Override
public void putInt(String key, int value) {
  cache.put(key, value);
}

@Override
public void putLong(String key, long value) {
  cache.put(key, value);
}

@Override
public void putFloat(String key, float value) {
  cache.put(key, value);
}

@Override
public void putBoolean(String key, boolean value) {
  cache.put(key, value);
}

@SuppressWarnings("Since15")
@Override
public String getString(String key, String defValue) {
  return((String)cache.getOrDefault(key, defValue));
}

@SuppressWarnings("Since15")
@Override
public Set<String> getStringSet(String key, Set<String> defValues) {
```

```
    return((Set<String>)cache.getOrDefault(key, defValues));
  }

  @SuppressWarnings("Since15")
  @Override
  public int getInt(String key, int defValue) {
    return((Integer)cache.getOrDefault(key, defValue));
  }

  @SuppressWarnings("Since15")
  @Override
  public long getLong(String key, long defValue) {
    return((Long)cache.getOrDefault(key, defValue));
  }

  @SuppressWarnings("Since15")
  @Override
  public float getFloat(String key, float defValue) {
    return((Float)cache.getOrDefault(key, defValue));
  }

  @SuppressWarnings("Since15")
  @Override
  public boolean getBoolean(String key, boolean defValue) {
    return((Boolean)cache.getOrDefault(key, defValue));
  }
}
```

(from Prefs/DataStore/app/src/main/java/com/commonsware/android/preffrag/SillyDataStore.java)

This implementation is truly silly, as it does no type checking and no persistence. It should be considered the bare minimum implementation of a `PreferenceDataStore`, though one that might be useful, instead of rolling a mock, in unit testing.

(note: the `@SuppressWarnings("Since15")` annotations are because this code uses the `getOrDefault()` method on `HashMap`, which was added in Java 8 and is new to Android 8.0)

There is a singleton instance of `SillyDataStore`. That way we can ensure that the same instance is used wherever we want it.

We apply that singleton in the `PreferenceFragment` subclass:

```
  public static class Prefs extends PreferenceFragment {
    @Override
```

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getPreferenceManager().getPreferenceDataStore()==null) {
      getPreferenceManager().setPreferenceDataStore(SillyDataStore.get());
    }

    addPreferencesFromResource(R.xml.preferences);
  }
}
```

(from Prefs/DataStore/app/src/main/java/com/commonsware/android/preffrag/EditPreferences.java)

Here, we check to see if there is a `PreferenceDataStore` associated with the `PreferenceManager` and, if not, we attach the `SillyDataStore` singleton. This causes the `SharedPreferences` used by this `PreferenceFragment` to use the `SillyDataStore` for storage, instead of the default XML-based persistence.

This works... somewhat. There are some problems.

First, a `PreferenceManager` is not a system service. Of note, each instance of our `PreferenceFragment` gets its own fresh `PreferenceManager`. This is why we need to check for, and set, the `PreferenceDataStore` on the `PreferenceManager` for the `PreferenceFragment` each time.

Second, we have no way of associating a `PreferenceDataStore` with the default `SharedPreferences` obtained from `PreferenceManager.getDefaultSharedPreferences()`. That will always use the standard XML backing store. The initial activity contents are in the form of a `PreferenceContentsFragment` that reads from the default `SharedPreferences` in `onResume()`. Normally, that would cause changes that we make via our `PreferenceFragment` to show up when the `PreferenceContentsFragment` is resumed. In this specific sample, that does not happen, as the `PreferenceContentsFragment` is using the default XML data store, and our data is really in the `SillyDataStore`.

This illustrates the fatal flaw of this system: you can never really use the `SharedPreferences`. You cannot create `PreferenceManager` instances yourself, as it has no `public` constructors. All of the code that gives you `SharedPreferences` objects back *other* than through a `PreferenceManager` has no way of associating a `PreferenceDataStore` with the `SharedPreferences`.

Instead of using `SharedPreferences`... you have to read and write from your `PreferenceDataStore` itself. This, in turn, loses everything that you normally associate with `SharedPreferences`, such as atomicity of updates and preference-change listeners.

Worse, since `PreferenceDataStore` has no API telling you when to persist changes, you would have to do that yourself... somehow.

These can be overcome, with a sufficiently-robust `PreferenceDataStore` implementation and lots of documentation. However, for casual use, other than perhaps for test mocks, `PreferenceDataStore` is not a well-engineered solution and probably should be avoided.

Custom Dialogs and Preferences

Android ships with a number of dialog classes for specific circumstances, like `DatePickerDialog` and `ProgressDialog`. Similarly, Android comes with a smattering of `Preference` classes for your `PreferenceActivity`, to accept text or selections from lists and so on.

However, there is plenty of room for improvement in both areas. As such, you may find the need to create your own custom dialog or preference class. This chapter will show you how that is done.

We start off by looking at creating a custom `AlertDialog`, not by using `AlertDialog.Builder`, but via a custom subclass. Then, we show how to create your own dialog-style `Preference`, where tapping on the preference pops up a dialog to allow the user to customize the preference value.

Prerequisites

Understanding this chapter requires that you have read the chapter on dialogs, along with the chapter on the preference system. Also, the samples here use the custom `ColorMixer View` described in another chapter.

Your Dialog, Chocolate-Covered

For your own application, the simplest way to create a custom `AlertDialog` is to use `AlertDialog.Builder`, as described in a previous chapter. You do not need to create any special subclass — just call methods on the `Builder`, then `show()` the resulting dialog.

However, if you want to create a reusable `AlertDialog`, this may become problematic. For example, where would this code to create the custom `AlertDialog` reside?

So, in some cases, you may wish to extend `AlertDialog` and supply the dialog's contents that way, which is how `TimePickerDialog` and others are implemented. Unfortunately, this technique is not well documented. This section will illustrate how to create such an `AlertDialog` subclass, as determined by looking at how the core Android team did it for their own dialogs.

The sample code is `ColorMixerDialog`, a dialog wrapping around the `ColorMixer` widget shown in a previous chapter. The implementation of `ColorMixerDialog` can be found in the CWAC-ColorMixer GitHub repository, as it is part of the CommonsWare Android Components.

Using this dialog works much like using `DatePickerDialog` or `TimePickerDialog`. You create an instance of `ColorMixerDialog`, supplying the initial color to show and a listener object to be notified of color changes. Then, call `show()` on the dialog. If the user makes a change and accepts the dialog, your listener will be informed.

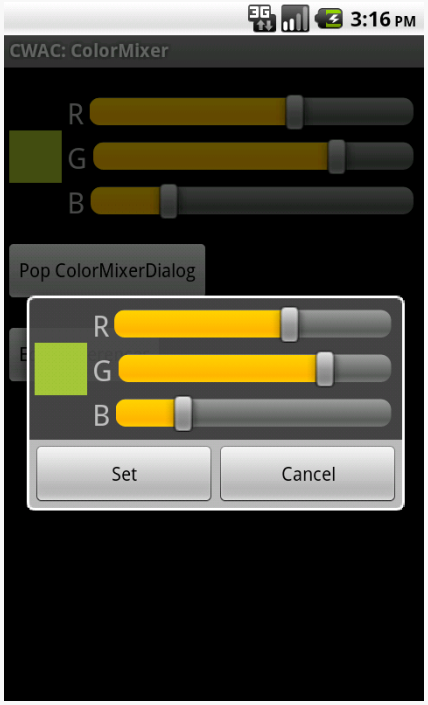


Figure 635: The ColorMixerDialog

Basic AlertDialog Setup

The `ColorMixerDialog` class is not especially long, since all of the actual color mixing is handled by the `ColorMixer` widget:

```
package com.commonsware.cwac.colormixer;

import android.app.AlertDialog;
import android.content.Context;
import android.content.DialogInterface;
import android.os.Bundle;

public class ColorMixerDialog extends AlertDialog
  implements DialogInterface.OnClickListener {
  static private final String COLOR="c";
  private ColorMixer mixer=null;
  private int initialColor;
  private ColorMixer.OnColorChangedListener onSet=null;

  public ColorMixerDialog(Context ctxt,
                          int initialColor,
                          ColorMixer.OnColorChangedListener onSet) {
    super(ctxt);

    this.initialColor=initialColor;
    this.onSet=onSet;

    mixer=new ColorMixer(ctxt);
    mixer.setColor(initialColor);

    setView(mixer);
    setButton(ctxt.getText(R.string.cwac_colormixer_set),
              this);
    setButton2(ctxt.getText(R.string.cwac_colormixer_cancel),
               (DialogInterface.OnClickListener)null);
  }

  @Override
  public void onClick(DialogInterface dialog, int which) {
    if (initialColor!=mixer.getColor()) {
      onSet.onColorChange(mixer.getColor());
    }
  }

  @Override
  public Bundle onSaveInstanceState() {
    Bundle state=super.onSaveInstanceState();
```

```
    state.putInt(COLOR, mixer.getColor());

    return(state);
  }

  @Override
  public void onRestoreInstanceState(Bundle state) {
    super.onRestoreInstanceState(state);

    mixer.setColor(state.getInt(COLOR));
  }
}
```

We extend the `AlertDialog` class and implement a constructor of our own design. In this case, we take in three parameters:

1. A `Context` (typically an `Activity`), needed for the superclass
2. The initial color to use for the dialog, such as if the user is editing a color they chose before
3. A `ColorMixer.OnColorChangedListener` object, just like `ColorMixer` uses, to notify the dialog creator when the color is changed

We then create a `ColorMixer` and call `setView()` to make that be the main content of the dialog. We also call `setButton()` and `setButton2()` to specify a "Set" and "Cancel" button for the dialog. The latter just dismisses the dialog, so we need no event handler. The former we route back to the `ColorMixerDialog` itself, which implements the `DialogInterface.OnClickListener` interface.

Handling Color Changes

When the user clicks the "Set" button, we want to notify the application about the color change...if the color actually changed. This is akin to `DatePickerDialog` and `TimePickerDialog` only notifying you of date or times if the user clicks Set and actually changed the values.

The `ColorMixerDialog` tracks the initial color via the `initialColor` data member. In the `onClick()` method — required by `DialogInterface.OnClickListener` — we see if the mixer has a different color than the `initialColor`, and if so, we call the supplied `ColorMixer.OnColorChangedListener` callback object:

```
  @Override
  public void onClick(DialogInterface dialog, int which) {
```

```
    if (initialColor!=mixer.getColor()) {
      onSet.onColorChange(mixer.getColor());
    }
  }
```

State Management

Dialogs use `onSaveInstanceState()` and `onRestoreInstanceState()`, just like activities do. That way, if the screen is rotated, or if the hosting activity is being evicted from RAM when it is not in the foreground, the dialog can save its state, then get it back later as needed.

The biggest difference with `onSaveInstanceState()` for a dialog is that the `Bundle` of state data is not passed into the method. Rather, you get the `Bundle` by chaining to the superclass, then adding your data to the `Bundle` it returned, before returning it yourself:

```
@Override
public Bundle onSaveInstanceState() {
  Bundle state=super.onSaveInstanceState();

  state.putInt(COLOR, mixer.getColor());

  return(state);
}
```

The `onRestoreInstanceState()` pattern is much closer to the implementation you would find in an `Activity`, where the `Bundle` with the state data to restore is passed in as a parameter:

```
@Override
public void onRestoreInstanceState(Bundle state) {
  super.onRestoreInstanceState(state);

  mixer.setColor(state.getInt(COLOR));
}
```

Preferring Your Own Preferences, Preferably

The Android Settings application, built using the `Preference` system, has lots of custom `Preference` classes. You too can create your own `Preference` classes, to collect things like dates, numbers, or colors. Once again, though, the process of creating such classes is not well documented. This section reviews one recipe for

making a `Preference` — specifically, a subclass of `DialogPreference` – based on the implementation of other `Preference` classes in Android.

The result is `ColorPreference`, a `Preference` that uses the `ColorMixer` widget. As with the `ColorMixerDialog` from the previous section, the `ColorPreference` is from the CommonsWare Android Components, and its source code can be found in the CWAC-ColorMixer GitHub repository.

One might think that `ColorPreference`, as a subclass of `DialogPreference`, might use `ColorMixerDialog`. However, that is not the way it works, as you will see.

The Constructor

A Preference is much like a custom `View`, in that there are a variety of constructors, some taking an `AttributeSet` (for the preference properties), and some taking a default style. In the case of `ColorPreference`, we need to get the string resources to use for the names of the buttons in the dialog box, providing them to `DialogPreference` via `setPositiveButtonText()` and `setNegativeButtonText()`.

Here, we just implement the standard two-parameter constructor, since that is the one that is used when this preference is inflated from a preference XML file:

```
public ColorPreference(Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  setPositiveButtonText(ctxt.getText(R.string.cwac_colormixer_set));
  setNegativeButtonText(ctxt.getText(R.string.cwac_colormixer_cancel));
}
```

Creating the View

The `DialogPreference` class handles the pop-up dialog that appears when the preference is clicked upon by the user. Subclasses get to provide the `View` that goes inside the dialog. This is handled a bit reminiscent of a `CursorAdapter`, in that there are two separate methods to be overridden:

- `onCreateDialogView()` works like `newView()` of `CursorAdapter`, returning a `View` that should go in the dialog
- `onBindDialogView()` works like `bindView()` of `CursorAdapter`, where the custom `Preference` is supposed to configure the `View` for the current preference value

In the case of `ColorPreference`, we use a `ColorMixer` for the `View`:

```
@Override
protected View onCreateDialogView() {
  mixer=new ColorMixer(getContext());

  return(mixer);
}
```

Then, in `onBindDialogView()`, we set the mixer's color to be `lastColor`, a private data member:

```
@Override
protected void onBindDialogView(View v) {
  super.onBindDialogView(v);

  mixer.setColor(lastColor);
}
```

We will see later in this section where `lastColor` comes from – for the moment, take it on faith that it holds the user's chosen color, or a default value.

Dealing with Preference Values

Of course, the whole point behind a `Preference` is to allow the user to set some value that the application will then use later on. Dealing with values is a bit tricky with `DialogPreference`, but not too bad.

Getting the Default Value

The preference XML format has an `android:defaultValue` attribute, which holds the default value to be used by the preference. Of course, the actual data type of the value will differ widely — an `EditTextPreference` might expect a `String`, while `ColorPreference` needs a color value.

Hence, you need to implement `onGetDefaultValue()`. This is passed a `TypedArray` — similar to how a custom `View` uses a `TypedArray` for getting at its custom attributes in an XML layout file. It is also passed an index number into the array representing `android:defaultValue`. The custom `Preference` needs to return an `Object` representing its interpretation of the default value.

In the case of `ColorPreference`, we simply get an integer out of the `TypedArray`, representing the color value, with an overall default value of `0xFFA4C639` (a.k.a., Android green):

```
@Override
protected Object onGetDefaultValue(TypedArray a, int index) {
  return(a.getInt(index, 0xFFA4C639));
}
```

Setting the Initial Value

When the user clicks on the preference, the `DialogPreference` supplies the last-known preference value to its subclass, or the default value if this preference has not been set by the user to date.

The way this works is that the custom `Preference` needs to override `onSetInitialValue()`. This is passed in a boolean flag (`restoreValue`) indicating whether or not the user set the value of the preference before. It is also passed the `Object` returned by `onGetDefaultValue()`. Typically, a custom `Preference` will look at the flag and choose to either use the default value or load the already-set preference value.

To get the existing value, `Preference` defines a set of type-specific getter methods — `getPersistedInt()`, `getPersistedString()`, etc. So, `ColorPreference` uses `getPersistedInt()` to get the saved color value:

```
@Override
protected void onSetInitialValue(boolean restoreValue, Object defaultValue) {
  lastColor=(restoreValue ? getPersistedInt(lastColor) : (Integer)defaultValue);
}
```

Here, `onSetInitialValue()` stores that value in `lastColor` — which then winds up being used by `onBindDialogView()` to tell the `ColorMixer` what color to show.

Closing the Dialog

When the user closes the dialog, it is time to persist the chosen color from the `ColorMixer`. This is handled by the `onDialogClosed()` callback method on your custom `Preference`:

```
@Override
protected void onDialogClosed(boolean positiveResult) {
  super.onDialogClosed(positiveResult);
```

```
    if (positiveResult) {
      if (callChangeListener(mixer.getColor())) {
        lastColor=mixer.getColor();
        persistInt(lastColor);
      }
    }
  }
```

The passed-in boolean indicates if the user accepted or dismissed the dialog, so you can elect to skip saving anything if the user dismissed the dialog. The other `DialogPreference` implementations also call `callChangeListener()`, which is somewhat ill-documented. Assuming both the flag and `callChangeListener()` are true, the `Preference` should save its value to the persistent store via `persistInt()`, `persistString()`, or kin.

Using the Preference

Given all of that, using the custom `Preference` class in an application is almost anti-climactic. You simply add it to your preference XML, with a fully-qualified class name:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <com.commonsware.cwac.colormixer.ColorPreference
    android:key="favoriteColor"
    android:defaultValue="0xFFA4C639"
    android:title="Your Favorite Color"
    android:summary="Blue.  No yel--  Auuuuuuuugh!" />
</PreferenceScreen>
```

At this point, it behaves no differently than does any other `Preference` type. Since `ColorPreference` stores the value as an integer, your code would use `getInt()` on the `SharedPreferences` to retrieve the value when needed.

The user sees an ordinary preference entry in the `PreferenceActivity`:

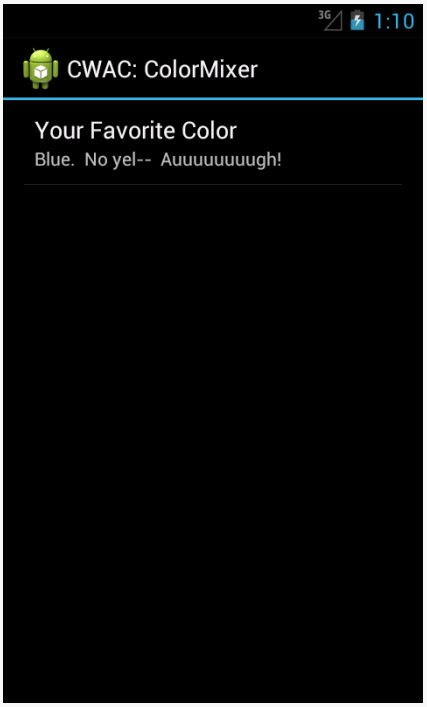


Figure 636: A PreferenceActivity, showing the ColorPreference

When tapped, it brings up the mixer:

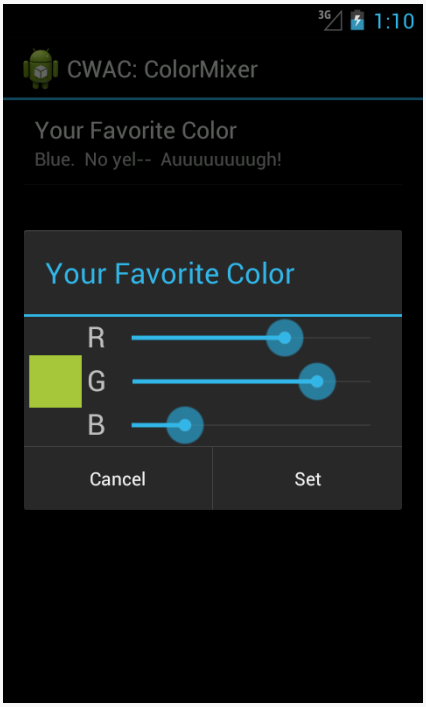


Figure 637: The ColorMixer in a custom DialogPreference

Choosing a color and clicking “Set” persists the color value as a preference.

Progress Indicators

Sometimes, we make the user wait. And wait. And wait some more.

Often, in these cases, it is useful to let the user know that something they requested is something that we are diligently working on. To do this, we can use some form of progress indicator. We saw basic use of a `ProgressBar` in the tutorials earlier in this book — now is the time to take a much closer look at `ProgressBar` and other means of displaying progress.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having read the chapters on dialogs, custom drawables, and animators is also a good idea.

Progress Bars

The classic way to tell the user that we are doing something for them is to use a `ProgressBar` widget, much as we briefly displayed one in the EmPubLite sample app in the tutorials.

However, a `ProgressBar` is much more than a simple spinning image. We can use it to display either indeterminate progress ("we will be done... sometime") or specific progress ("we are 34% complete"). We can use it either as a circle or as a classic horizontal bar, the latter typically used for specific progress. And, for specific progress, we can actually show two tiers of progress, known as "primary" and "secondary" (e.g., primary for the progress in copying a directory's worth of files, secondary for the progress on a specific file).

In this section, we will take a look at these different ways of using `ProgressBar`.

Circular vs. Horizontal

As the name suggests, a `ProgressBar` denotes progress. As the name does not suggest, a `ProgressBar` is not a bar, by default — it is a circle. Hence, the following element from an XML layout resource:

```
<ProgressBar
  android:id="@+id/progressCI"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_gravity="center_horizontal"
  android:layout_marginBottom="20dp"
  android:layout_marginTop="20dp"/>
```

(from Progress/BarSampler/app/src/main/res/layout/activity_main.xml)

gives us:

Figure 638: Android 5.1 ProgressBar, Default Style

Figure 639: Android 4.0 ProgressBar, Default Style

However, referencing `style="?android:attr/progressBarStyleHorizontal"` in the element:

```
<ProgressBar
  android:id="@+id/progressHI"
  style="?android:attr/progressBarStyleHorizontal"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_marginBottom="20dp"
  android:indeterminate="true"/>
```

(from Progress/BarSampler/app/src/main/res/layout/activity_main.xml)

gives us a horizontal bar:

Figure 640: Android 5.1 ProgressBar, Horizontal Style

Figure 641: Android 4.0 ProgressBar, Horizontal Style

Note that the look-and-feel of these widgets have changed over the years. On Android 1.x and 2.x, they will look like this:

Figure 642: Android 2.3.3 ProgressBar, Both Styles

Specific vs. Indeterminate

Typically, you use the circular `ProgressBar` style for indeterminate progress, where the circle simply spins in place to let the user know that work is proceeding and the device (or activity) has not frozen. The horizontal `ProgressBar` style is used to illustrate specific amounts of progress, from 0 to a value you choose.

However, while those patterns are typical, the choice of whether to use indeterminate or some specific amount of progress is independent of the style of the widget.

The `android:indeterminate` attribute controls whether the `ProgressBar` will render an indeterminate look or a specific look. For the latter, calls to `setMax()` (or the `android:max` attribute) will set the upper end of the progress range (the default is 100), and `setProgress()` or `incrementProgressBy()` will set how much progress along that range is illustrated.

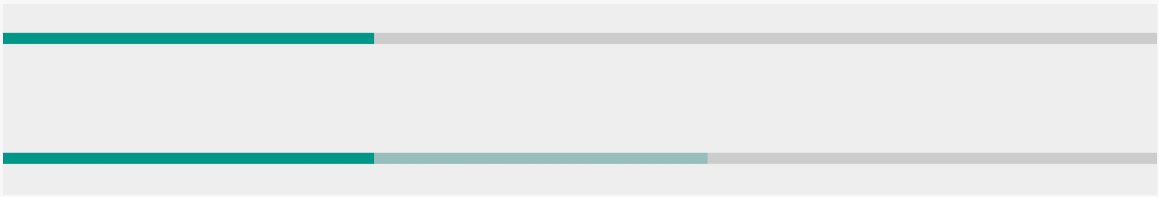


Figure 643: Android 5.1 ProgressBar, Horizontal Style, Indeterminate and Specific

Figure 644: Android 4.0 ProgressBar, Horizontal Style, Indeterminate and Specific

Figure 645: Android 2.3.3 ProgressBar, Horizontal Style, Indeterminate and Specific

Primary vs. Secondary

For specific progress, you actually have two independent amounts of progress. `setProgress()`, `incrementProgressBy()`, and `android:progress` control the primary progress, while `setSecondaryProgress()`, `incrementSecondaryProgressBy()`, and `android:secondaryProgress` control the secondary progress. Here, "primary progress" refers to the progress along an entire piece of work (e.g., copying a folder's worth of files), while "secondary progress" refers the progress along a discrete chunk of the overall work (e.g., copying an individual file).

A `ProgressBar` will render these with different colors, though primary trumps secondary, and so the secondary progress will only be visible when its value exceeds that of the primary progress:

Figure 646: Android 4.0 ProgressBar, Horizontal Style, Primary-Only and Primary-Plus-Secondary

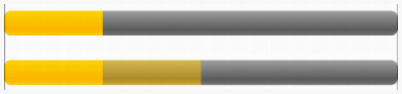


Figure 647: Android 2.3.3 ProgressBar, Horizontal Style, Primary-Only and Primary-Plus-Secondary

ProgressBar and Threads

Normally, you cannot update the UI of a widget from a background thread.

`ProgressBar` is an exception. You *can* safely call `setProgress()` and `incrementProgressBy()` from a background thread to update the primary progress, and you can safely call `setSecondaryProgress()` and `incrementSecondaryProgressBy()` from a background thread to update the secondary progress.

To see this in action, take a look at the `Progress/BarSampler` sample project.

This project has a single activity (`MainActivity`), whose layout (`activity_main.xml`) contains four `ProgressBar` widgets, two indeterminate and two for specific progress:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ProgressBar
    android:id="@+id/progressCI"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginBottom="20dp"
    android:layout_marginTop="20dp"/>

  <ProgressBar
    android:id="@+id/progressHI"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:indeterminate="true"/>

  <ProgressBar
    android:id="@+id/progressHS"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:indeterminate="false"
    android:max="100"/>

  <ProgressBar
    android:id="@+id/progressHS2"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:max="100"/>

</LinearLayout>
```

(from Progress/BarSampler/app/src/main/res/layout/activity_main.xml)

The activity gets access to the latter two `ProgressBar` widgets and sets up a `ScheduledThreadPoolExecutor` to get control every second in a background thread, which calls our `run()` method. The `run()` method will increment both `ProgressBar` widgets primary progress by 2 each time, and the secondary progress by 10 (dropping back to the starting point when the secondary progress reaches the

maximum of 100). When the primary progress gets to 100, we cancel our scheduled work in the `ScheduledThreadPoolExecutor`:

```
package com.commonsware.android.progress;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ProgressBar;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MainActivity extends Activity implements Runnable {
  private static final int PERIOD_SECONDS=1;
  private ScheduledThreadPoolExecutor executor=
      new ScheduledThreadPoolExecutor(1);
  private ProgressBar primary=null;
  private ProgressBar secondary=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    primary=(ProgressBar)findViewById(R.id.progressHS);
    secondary=(ProgressBar)findViewById(R.id.progressHS2);

    executor.setExecuteExistingDelayedTasksAfterShutdownPolicy(false);
    executor.scheduleAtFixedRate(this, 0, PERIOD_SECONDS,
                                 TimeUnit.SECONDS);
  }

  @Override
  public void onDestroy() {
    executor.shutdown();

    super.onDestroy();
  }

  @Override
  public void run() {
    if (primary.getProgress() < 100) {
      primary.incrementProgressBy(2);
      secondary.incrementProgressBy(2);

      if (secondary.getSecondaryProgress() == 100) {
        secondary.setSecondaryProgress(10);
      }
```

```
      else {
        secondary.incrementSecondaryProgressBy(10);
      }
    }
    else {
      executor.remove(this);
    }
  }
}
```

(from Progress/BarSampler/app/src/main/java/com/commonsware/android/progress/MainActivity.java)

The net effect is that you see the progress march across the screen, with the secondary progress going through five passes for the primary progress' single pass through the 0-100 range.

Tailoring Progress Bars

The stock `ProgressBar` look and feel is decent, if perhaps not spectacular. Often times, the stock look is sufficient for your needs. If you wish to have greater control over the look of your `ProgressBar`, the following sections will demonstrate some possibilities.

Changing the Progress Colors

The `ProgressBar` uses different colors for primary and secondary specific progress. By default, those colors are defined by the theme you are using, and the stock themes have firmware-defined colors (e.g., yellows for Android 1.x and 2.x, blues for Android 3.x and higher).

However, you can change the colors by using a `LayerListDrawable` and associating it with a `ProgressBar` by means of the `android:progressDrawable` attribute.

The `ProgressBar` background image needs to be a `LayerListDrawable` with three specific layers:

- `android:id="@android:id/background"` for the background color of the bar
- `android:id="@android:id/progress"` for the primary progress
- `android:id="@android:id/secondaryProgress"` for the secondary progress

Whether those layers are defined as `ShapeDrawable` structures, or as nine-patch PNG files is up to you, but they will need the ability to stretch to fit however big your bar winds up being.

To see what this means, let's take a look at the `Progress/Styled` sample project. This is a near-clone of the `Progress/BarSampler` project from earlier, using custom backgrounds for the bars. Here, we will look at the horizontal `ProgressBar` widgets — in the next section, we will look at how to change the background of a circular indefinite `ProgressBar`.

For the first horizontal `ProgressBar` (`progressHS`), for Android 4.x, we will use a custom style created by the Android Holo Colors Generator, a Web site set up to help us create custom versions of the holographic widget theme.

When you visit this site in Google Chrome (note: other browsers are not supported at this time), you can fill in a name for your theme (e.g., "AppTheme"), the color scheme to use for the theme, and the foundation theme to use (light or dark):

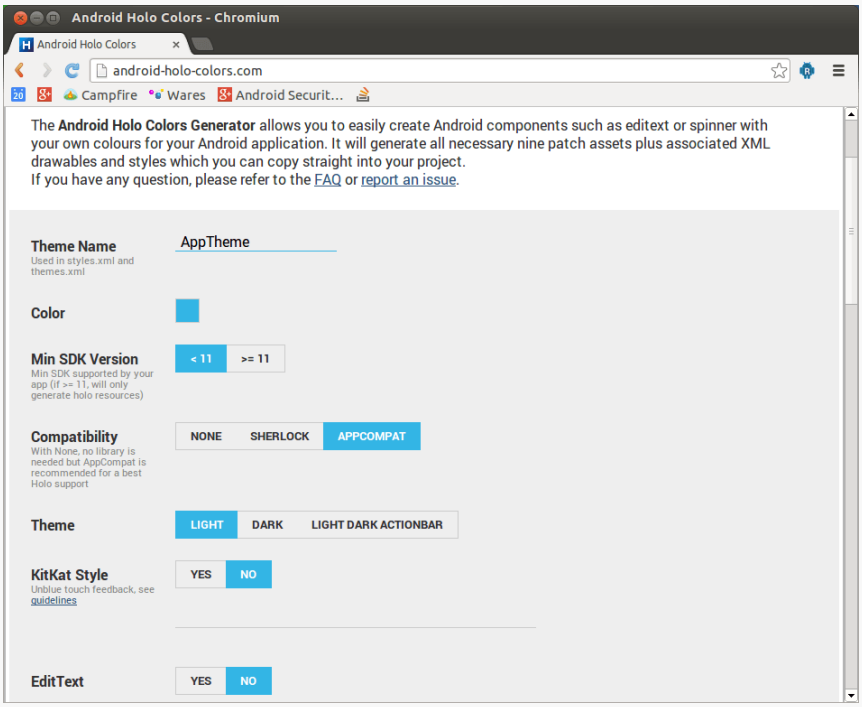


Figure 648: Android Holo Colors Generator, Basic Info

You can then toggle on and off which widgets you intend to use, so the generator will create custom styles for them:

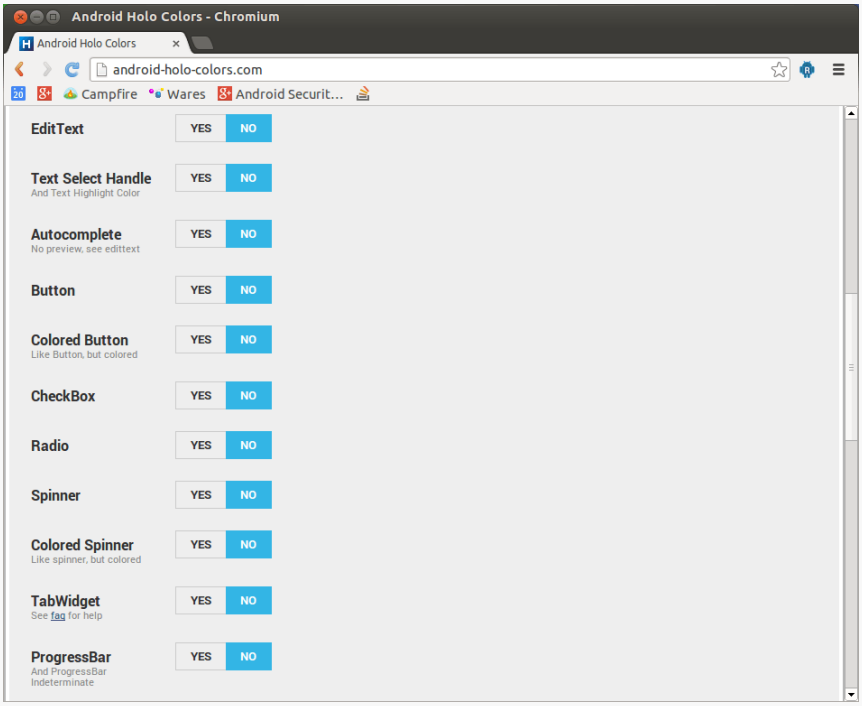


Figure 649: Android Holo Colors Generator, Widget Selection

Then, the generator will create a ZIP file that you can download that contains the generated resources for your custom styles.

The `Progress/Styled` project contains the files generated by the generator, replacing the original style resources. Note that the generator does not create a `.DarkActionBar` version of the style resource, so the `values-v14` resource directory in the project has one hand-crafted based upon a regular generated style resource.

On Android 5.0+, we will use a theme with the same name, but where we are using tints in the theme to affect the colors of the progress indicators.

Our manifest points to our `AppTheme` as being how we wish to style widgets in this application:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.progress"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="8"
```

```
    android:targetSdkVersion="15" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/title_activity_main">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Progress/Styled/app/src/main/AndroidManifest.xml)

That theme, defined in `apptheme_themes.xml`, points to style resources for horizontal `ProgressBar` widgets:

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Generated (in part) with http://android-holo-colors.com -->
<resources xmlns:android="http://schemas.android.com/apk/res/android">

  <style name="AppTheme" parent="android:Theme.Holo.Light.DarkActionBar">

    <item name="android:progressBarStyleHorizontal">@style/ProgressBarAppTheme</item>

  </style>

</resources>
```

(from Progress/Styled/app/src/main/res/values-v14/apptheme_themes.xml)

The `ProgressBarAppTheme` style resource is defined in a separate `apptheme_styles.xml` resource:

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Generated with http://android-holo-colors.com -->
<resources xmlns:android="http://schemas.android.com/apk/res/android">

  <style name="ProgressBarAppTheme" parent="android:Widget.Holo.Light.ProgressBar.Horizontal">
      <item name="android:progressDrawable">@drawable/progress_horizontal_holo_light</item>
      <item name="android:indeterminateDrawable">@drawable/
progress_indeterminate_horizontal_holo_light</item>
  </style>
```

```
</resources>
```

(from Progress/Styled/app/src/main/res/values-v11/apptheme_styles.xml)

Here, we say that we want the `android:progressDrawable` property to be a `progress_horizontal_holo_light` drawable resource. We also set the `android:indeterminateDrawable` property — used for indeterminate bars — to a `progress_indeterminate_horizontal_holo_light` drawable resource.

Those are defined as XML-based drawables, in the `res/drawable/` directory in the project. The `progress_horizontal_holo_light` resource is defined as:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2010 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@android:id/background"
          android:drawable="@drawable/progress_bg_holo_light" />

    <item android:id="@android:id/secondaryProgress">
        <scale android:scaleWidth="100%"
               android:drawable="@drawable/progress_secondary_holo" />
    </item>

    <item android:id="@android:id/progress">
        <scale android:scaleWidth="100%"
               android:drawable="@drawable/progress_primary_holo" />
    </item>

</layer-list>
```

(from Progress/Styled/app/src/main/res/drawable/progress_horizontal_holo_light.xml)

The generator creates our `LayerListDrawable` resource with our three layers, each pointing to a nine-patch PNG file (with different versions for different densities) that contains our desired custom color. The `progress` and `secondaryProgress` layers use `ScaleDrawable` definitions to ensure that the images are measured against the complete width of the background layer, which in turn will be sized according to the size of the `ProgressBar` itself.

We will take a look at the `progress_indeterminate_horizontal_holo_light` drawable resource in the next section.

Note that you could skip the custom theme and style if you wished, and simply add the `android:progressDrawable` attribute to the `ProgressBar` widget definition in its layout XML resource.

Regardless, the result is that our progress bars have the desired purple color scheme:

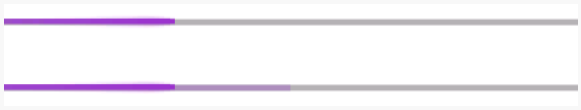


Figure 650: Custom ProgressBar Style, Primary and Secondary

Also, you can have your `LayerListDrawable` use `ShapeDrawable` layers, to avoid creating nine-patch PNG files, if you prefer, using a resource like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/app/src/main/res/android">
    <item android:id="@android:id/background">
        <shape>
            <stroke android:width="1dip" android:color="#FF333333" />
            <gradient
                    android:startColor="#FF9C9E9C"
                    android:centerColor="#FF5A5D5A"
                    android:centerY="0.71"
                    android:endColor="#FF6B716B"
                    android:angle="270"
            />
        </shape>
    </item>
    <item android:id="@android:id/secondaryProgress">
        <clip>
            <shape>
            <stroke android:width="1dip" android:color="#FF333333" />
                <gradient
                        android:startColor="#4cffffff"
```

```
                        android:centerColor="#4cE7E7E7"
                        android:centerY="0.71"
                        android:endColor="#4cFFFBFF"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>
    <item android:id="@android:id/progress">
        <clip>
            <shape>
                <stroke android:width="1dip" android:color="#FF333333" />
                <gradient
                        android:startColor="#FFFFFFFF"
                        android:centerColor="#FFE7E7E7"
                        android:centerY="0.71"
                        android:endColor="#FFFFFBFF"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>
</layer-list>
```

On Android 5.0+, we have it much easier, as `ProgressBar` automatically adopts the accent tint. So we go with a much simpler theme definition:

```
<resources>

  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>

</resources>
```

(from Progress/Styled/app/src/main/res/values-v21/styles.xml)

This references colors from a separate `colors.xml` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#3f51b5</color>
  <color name="primary_dark">#1a237e</color>
  <color name="accent">#ffee58</color>
</resources>
```

(from Progress/Styled/app/src/main/res/values-v21/colors.xml)

The result are yellow-tinted progress bars:

Figure 651: Material ProgressBar Style, Primary and Secondary

Changing the Indeterminate Animation

Similarly, for indefinite progress "bars", changing the progress drawable will let you change the way they look. However, in this case, the drawable also needs to implement the animation itself. You can accomplish this either by using an `AnimationDrawable` or by using some other type of drawable wrapped in an animation, such as a `ShapeDrawable` wrapped in a `<rotate>` animation.

For example, the Android 4.x custom theme created by the Android Holo Colors Generator assigns the following drawable resource to `android:indeterminateDrawable` in the theme:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/*
** Copyright 2011, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->
<animation-list
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/progressbar_indeterminate_holo1" android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo2" android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo3" android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo4" android:duration="50" />
```

```
    <item android:drawable="@drawable/progressbar_indeterminate_holo5" android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo6" android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo7" android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo8" android:duration="50" />
</animation-list>
```

(from Progress/Styled/app/src/main/res/drawable/progress_indeterminate_horizontal_holo_light.xml)

Hence, every horizontal indeterminate `ProgressBar` will use that `AnimationDrawable`. The individual images in the animation are PNG files, with different versions for different densities.

Circular `ProgressBar` widgets also need a custom progress drawable, though obviously the image will need to be circular, not a bar. You can certainly use an `AnimationDrawable` for this, or you can use a `ShapeDrawable`, such as the `res/drawable/progress_circular.xml` resource shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromDegrees="0"
  android:pivotX="50%"
  android:pivotY="50%"
  android:toDegrees="360">

  <shape
    android:innerRadiusRatio="3"
    android:shape="ring"
    android:thicknessRatio="8"
    android:useLevel="false">
    <gradient
      android:centerColor="#4c737373"
      android:centerY="0.50"
      android:endColor="#ff9933CC"
      android:startColor="#4c737373"
      android:type="sweep"
      android:useLevel="false"/>
  </shape>

</rotate>
```

(from Progress/Styled/app/src/main/res/drawable/progress_circular.xml)

Here, we have a `ring` `ShapeDrawable`, with a certain thickness and radius, filled with a gradient. Half of the fill is actually a solid color (`#4c737373`), as the start and center colors are the same. The other half is a sweep gradient from the starting color to the same purple shade that is used by the other bar styles. This ring is then wrapped in a

`rotate` animation. This yields a simple gradient-filled ring, that rotates smoothly to indicate progress:

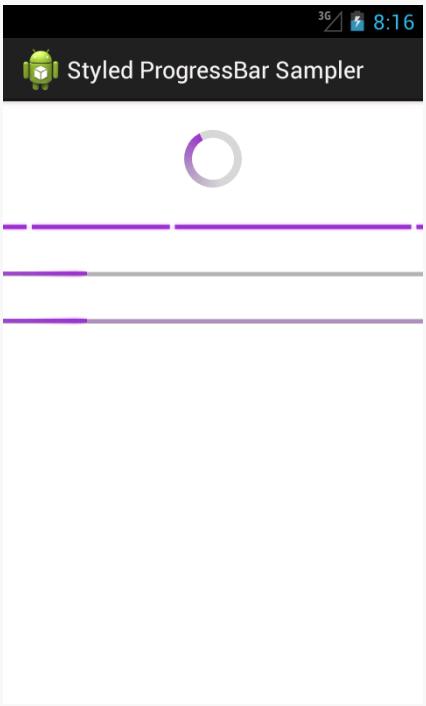


Figure 652: Custom ProgressBar Styles, Including Circular Indefinite

Note that the Android Holo Colors Generator does not generate circular indefinite `ProgressBar` resources as of the time of this writing.

Once again, Android 5.0+ can leverage `Theme.Material` and get rid of all the extra clutter. Just having an accent color defined will have your indefinite progress bars adopt that same color:

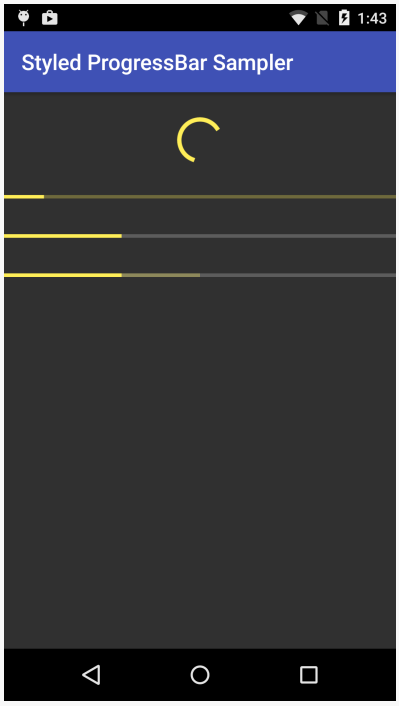


Figure 653: Material ProgressBar Styles, Including Circular Indefinite

Progress Dialogs

One use of a `ProgressBar` is to have it wrapped in a `ProgressDialog`. Like all dialogs, `ProgressDialog` is modal, preventing the user from interacting with an underlying activity while the dialog is displayed. From a UI design standpoint, a `ProgressDialog` is an easy way to temporarily show progress without having to find a spot for a `ProgressBar` widget somewhere in the UI. Also, since usually there are things in the activity that are dependent upon the work being done in the background, having the dialog in place prevents anyone from trying to use things that are not yet ready.

However, modal dialogs are not a great design approach, as they aggressively limit the user's options. `ProgressDialog` is perhaps the worst in this regard, as the user can do nothing except wait. While *part* of your app may not yet be ready, other parts surely are, such as reading the documentation, or adjusting settings, or clicking on your ad banners. Hence, using *anything* else other than `ProgressDialog`, while perhaps a bit more work, will be an improvement in the usability of your app.

That being said, let us see how to set up a `ProgressDialog`. The `Progress/Dialog` sample project is a near-clone of the `Dialogs/DialogFragment` sample project from the chapter on dialogs. The only difference is the `onCreateDialog()` method of our `DialogFragment`, where we directly create a `ProgressDialog` instead of using an `AlertDialog.Builder` to create an `AlertDialog` as before:

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
  ProgressDialog dlg=new ProgressDialog(getActivity());

  dlg.setMessage(getActivity().getString(R.string.dlg_title));
  dlg.setIndeterminate(true);
  dlg.setProgressStyle(ProgressDialog.STYLE_SPINNER);

  return(dlg);
}
```

(from Progress/Dialog/app/src/main/java/com/commonsware/android/progdlg/SampleDialogFragment.java)

We create the `ProgressDialog` via its constructor, set the message explaining what we are waiting for via `setMessage()`, indicate that the `ProgressBar` should be an indeterminate one via `setIndeterminate()`, and indicate that we want a circular "spinner" `ProgressBar` rather than a horizontal one by calling `setProgressStyle(ProgressDialog.STYLE_SPINNER)`. There are a variety of other things you could configure on the `ProgressDialog` if desired, and `ProgressDialog` inherits from `AlertDialog`, so some things you could configure on an `AlertDialog` will also be available on the `ProgressDialog`.

The result is a dialog that you may have seen from other apps in Android:

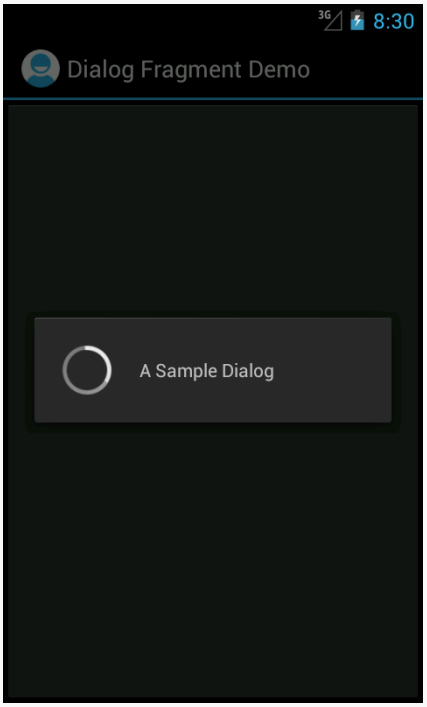


Figure 654: ProgressDialog

Title Bar and Action Bar Progress Indicators

Another place to let users know that you are doing something on their behalf is to put a progress indicator in the title bar or action bar of your activity. This avoids your having to put an indeterminate `ProgressBar` somewhere in your activity's UI. It is also very simple to set up, as we can see in the [Progress/TitleBar] sample project.

```
package com.commonsware.android.titleprog;

import android.app.Activity;
import android.os.Bundle;
import android.view.Window;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    getWindow().requestFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);
    setProgressBarIndeterminateVisibility(true);
```

```
  }
}
```

(from Progress/TitleBar/app/src/main/java/com/commonsware/android/titleprog/MainActivity.java)

Up front, as the first thing that you do in your `onCreate()` call, you need to call:

```
getWindow().requestFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
```

This tells Android to reserve space in your title bar or action bar for an indeterminate progress indicator, though the indicator does not appear at this point.

Later on, when you want the indicator to actually appear, call `setProgressBarIndeterminateVisibility(true)` on your activity, and later call `setProgressBarIndeterminateVisibility(false)` to make the indicator go away.

This particular application has `android:targetSdkVersion` set to `11` or higher, but it is not using an action bar backport. Hence, when you run it on an older Android environment, you get a classic title bar with the progress indicator on the right:

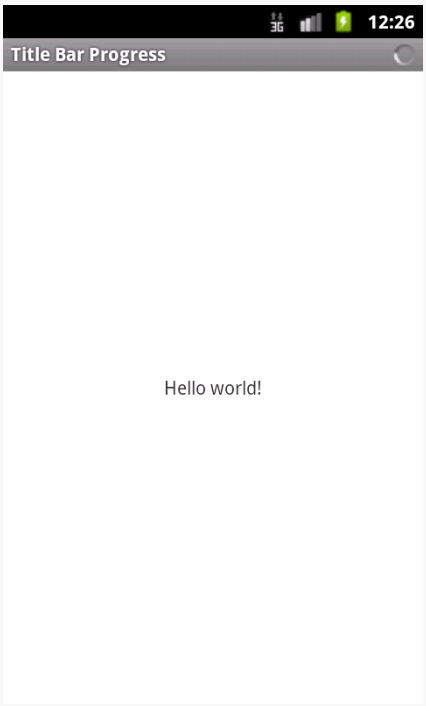


Figure 655: Progress Indicator in Title Bar

When you have an action bar, you get the same basic effect, albeit with a larger indicator to match the larger bar:

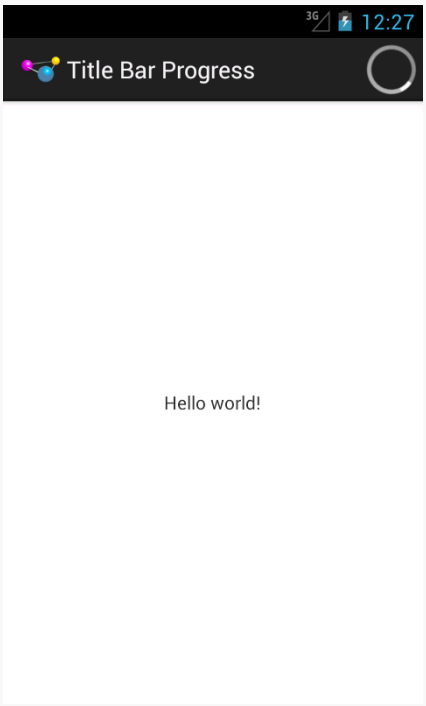


Figure 656: Progress Indicator in Action Bar

Note that this approach is not supported by `Theme.Material` or the `appcompat-v7` action bar backport, which makes it far less commonly used today.

Direct Progress Indication

Sometimes, the best way to let the user know about updates is to simply update the data in place. Rather than have some separate indicator, let the core UI itself convey the work being done.

We saw this in the chapter on threads, where we populated a `ListView` in "real time" as we loaded in data into its adapter. Other variations on this theme include:

- Updating a page count `TextView` to show the number of downloaded pages, while the user is reading earlier pages, perhaps with some sort of style (e.g., italics) or color coding (e.g., red) to indicate data that is being loaded.

- Simply disabling the buttons, action bar items, and other ways that the user could navigate to a point in your app where you need the data that is being loaded in the background. The key here is to make sure that users understand *why* those items are disabled, and sometimes that is not obvious. Hence, while this step may be necessary, it is often tied in with progress indicators in the title bar or action bar or other means of indicating to the user the *reason* they cannot perform certain operations.

More Fun with Pagers

In earlier chapters, we saw basic uses of `ViewPager`, along with ways to show multiple pages at a time on larger screens. However, there are other ways to apply `ViewPager` and integrate it into the rest of your application, some of which we will examine in this chapter.

Prerequisites

This chapter assumes that you have read the core chapters, particularly the one showing how to use `ViewPager`.

Hosting ViewPager in a Fragment

Classically, the primary restriction on `ViewPager` was that you could not both have `ViewPager` be in a fragment *and* have `ViewPager` host fragments as its pages. You could do one or the other, but not both simultaneously.

As noted in a previous chapter, Android 4.2+ supports nested fragments natively, and the latest Android Support package backport also supports nested fragments. With those, you can have `ViewPager` be in a fragment and host fragments as its pages. However, it requires a minor modification to the way we set up a `PagerAdapter`, as is illustrated in the `ViewPager/Nested` sample project. This is the same project as `ViewPager/Indicator`, with the twist that the pages are fragments and the `ViewPager` is inside a fragment.

Our activity now implements the standard add-the-fragment-if-it-does-not-exist pattern that we have seen previously:

```
package com.commonsware.android.pagernested;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class ViewPagerIndicatorActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
      getSupportFragmentManager().beginTransaction()
                                 .add(android.R.id.content,
                                      new PagerFragment()).commit();
    }
  }
}
```

(from ViewPager/Nested/app/src/main/java/com/commonsware/android/pagernested/ViewPagerIndicatorActivity.java)

This loads a `PagerFragment`, which contains most of the logic from our original activity:

```
package com.commonsware.android.pagernested;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class PagerFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.pager, container, false);
    ViewPager pager=(ViewPager)result.findViewById(R.id.pager);

    pager.setAdapter(buildAdapter());

    return(result);
  }

  private PagerAdapter buildAdapter() {
    return(new SampleAdapter(getActivity(), getChildFragmentManager()));
  }
}
```

(from ViewPager/Nested/app/src/main/java/com/commonsware/android/pagernested/PagerFragment.java)

The biggest difference is that our call to the constructor of `SampleAdapter` no longer uses `getSupportFragmentManager()`. Instead, it uses `getChildFragmentManager()`. This allows `SampleAdapter` to use fragments hosted by `PagerFragment`, rather than ones hosted by the activity as a whole.

No other code changes are required, and from the user's standpoint, there is no visible difference.

Pages and the Action Bar

Fragments that are pages inside a `ViewPager` can participate in the action bar, supplying items to appear as toolbar buttons, in the overflow menu, etc. This is not significantly different than how any fragment participates in the action bar:

- Call `setHasOptionsMenu()` early in the fragment lifecycle (e.g., `onCreateView()`) to state that the fragment wishes to contribute to the action bar contents
- Override `onCreateOptionsMenu()` and `onOptionsItemSelected()`, much as you would with an activity

`ViewPager` and `FragmentManager` will manage the contents of the action bar, based upon the currently-visible page. That page's contributions will appear in the action bar, then will be removed when the user switches to some other page.

To see this in action, take a look at the `ViewPager/ActionBar` sample project. This is the same as the `ViewPager/Indicator` project from before, except:

- In `onCreateView()`, for even-numbered page positions (0, 2, etc.), we call `setHasOptionsMenu(true)`:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);
  EditText editor=(EditText)result.findViewById(R.id.editor);

  position=getArguments().getInt(KEY_POSITION, -1);
  editor.setHint(getTitle(getActivity(), position));
```

```
  if ((position % 2)==0) {
    setHasOptionsMenu(true);
  }

  return(result);
}
```

(from ViewPager/ActionBar/app/src/main/java/com/commonsware/android/pagerbar/EditorFragment.java)

- In `onCreateOptionsMenu()`, we inflate a `res/menu/actions.xml` menu resource:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.actions, menu);

  super.onCreateOptionsMenu(menu, inflater);
}
```

(from ViewPager/ActionBar/app/src/main/java/com/commonsware/android/pagerbar/EditorFragment.java)

Normally, we would also implement `onOptionsItemSelected()`, to find out when the action bar item was tapped, though this is skipped in this sample.

The result is that when we have an even-numbered page position — equating to an odd-numbered title and hint — we have items in the action bar:

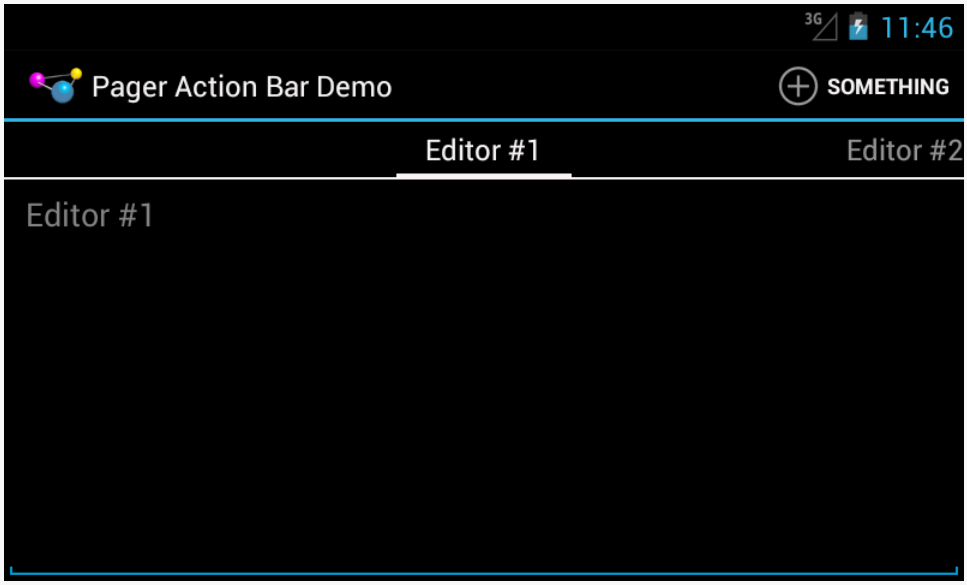


Figure 657: A ViewPager, PagerTabStrip, and Action Bar Item on Android 4.1

...but as soon as we swipe to an odd-numbered page position — equating to an even-numbered title and hint — our action bar item is removed, as that fragment did not call `setHasOptionsMenu(true)`:

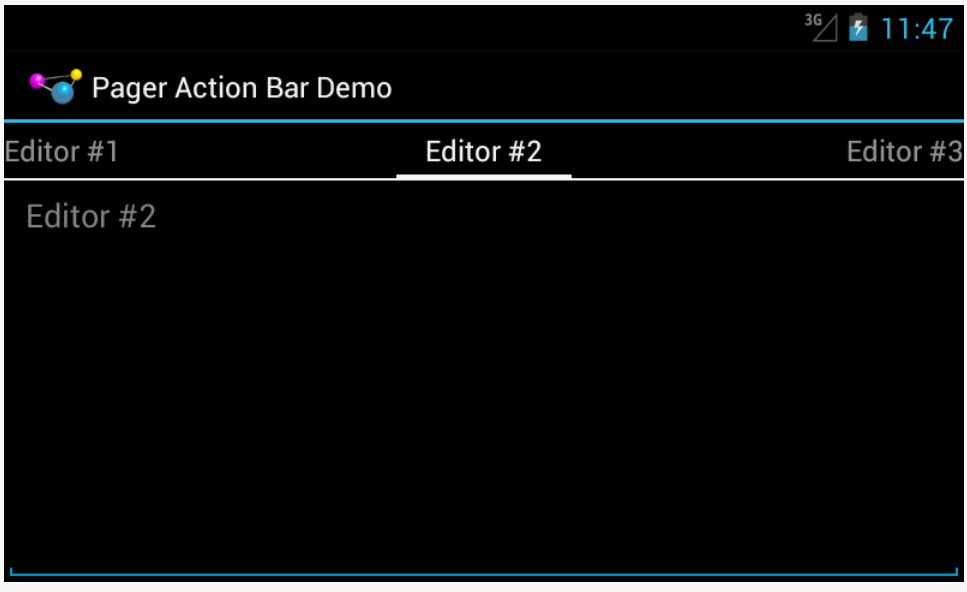


Figure 658: A ViewPager and PagerTabStrip, Sans Action Bar Item on Android 4.1

ViewPagers and Scrollable Contents

There are other things in Android that can be scrolled horizontally, besides a `ViewPager`:

- `HorizontalScrollView`
- `WebView`, for content that is wider than the width of the screen
- the deprecated `Gallery` widget
- maps from many mapping engines, such as Google Maps
- various third-party widgets

The challenge then comes in terms of dealing with horizontal swipe events. The ideal situation is for you to be able to swipe horizontally on the material inside the page, until you hit some edge (e.g., end of the `HorizontalScrollView`), then have swipe events move you to the adjacent page.

You can assist `ViewPager` in handling this scenario by subclassing it and overriding the `canScroll()` method. This will be called on a horizontal swipe, and it is up to you to indicate if the contents can be scrolled (returning `true`) or not (returning `false`). If the built-in logic is insufficient, tailoring `canScroll()` to your particular needs can help.

We will see an example of this later in the book, when we put some maps into a `ViewPager`.

Columns for Large, Pages for Small

In some cases, you can take better advantage of larger screens by using `ViewPager` more judiciously. In a previous chapter, we explored having `ViewPager` itself display more than one page at a time. A variation on that same theme is to only use a `ViewPager` on screen sizes where you lack sufficient room for everything, and to put those same pages on the screen at the same time when you have room for all of them.

For example, a Twitter client for Android could use the columns-or-pages support for displaying various streams of tweets: your timeline, your @ mentions, hashtags you follow, etc. Each stream is represented by a typical `ListView`, with one row per tweet. On a phone, since screen space is at a premium, those `ListView` widgets are set up in a `ViewPager`, with one list per page. Users can swipe between the lists, or use tabs to navigate the available lists. However, tablets offer more room, so the app

could show three `ListView` widgets side-by-side in landscape mode, so you can take in three sets of content without further interaction with the screen.

The `ViewPager/Columns1` sample project will demonstrate how you can accomplish the same basic approach in your own app... with some limitations.

The Layouts

Our main activity layout — cunningly named `main` — has a `ViewPager`-based definition in `res/layout/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager android:id="@+id/pager"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>

</android.support.v4.view.ViewPager>
```

(from ViewPager/Columns1/app/src/main/res/layout/main.xml)

However, in `res/layout-large/`, for 5-inch devices on up, we have a horizontal `LinearLayout` with three `FrameLayout` containers, each representing an equal-sized slot for one of our "pages":

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:baselineAligned="false"
  android:orientation="horizontal">

  <FrameLayout
    android:id="@+id/editor1"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

  <FrameLayout
    android:id="@+id/editor2"
```

```
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

  <FrameLayout
    android:id="@+id/editor3"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

</LinearLayout>
```

(from ViewPager/Columns1/app/src/main/res/layout-large/main.xml)

Android will automatically inflate the proper layout when we call `setContentView(R.layout.main)`.

The Activity

However, while Android handles the inflation for us, we obviously need to populate the contents a bit differently. In this sample, though, we are relying upon the fact that screen size will not change on the fly. Hence, an instance of our application will either show a `ViewPager` *or* show the horizontal `LinearLayout`, and not have to switch between those at runtime.

Our `SampleAdapter`, therefore, can remain unchanged, except for reducing the page count to 3:

```
package com.commonsware.android.pagercolumns;

import android.content.Context;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  Context ctxt=null;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(3);
```

```
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }

  @Override
  public String getPageTitle(int position) {
    return(EditorFragment.getTitle(ctxt, position));
  }
}
```

(from ViewPager/Columns1/app/src/main/java/com/commonsware/android/pagercolumns/SampleAdapter.java)

Our `MainActivity` will still use the `SampleAdapter`, and if we have a `ViewPager`, it will use it the same way as before. However, if we do *not* have a `ViewPager`, we must be showing three panes of content side by side, in which case we just execute a `FragmentTransaction` to populate the three `FrameLayout` containers with the three items created by the `SampleAdapter`:

```
package com.commonsware.android.pagercolumns;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentPagerAdapter;
import android.support.v4.view.ViewPager;

public class MainActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=findViewById(R.id.pager);

    if (pager==null) {
      if (getSupportFragmentManager().findFragmentById(R.id.editor1)==null) {
        FragmentPagerAdapter adapter=buildAdapter();

        getSupportFragmentManager().beginTransaction()
                                   .add(R.id.editor1,
                                        adapter.getItem(0))
                                   .add(R.id.editor2,
                                        adapter.getItem(1))
                                   .add(R.id.editor3,
                                        adapter.getItem(2)).commit();
```

```
      }
    }
    else {
      pager.setAdapter(buildAdapter());
    }
  }

  private FragmentPagerAdapter buildAdapter() {
    return(new SampleAdapter(this, getSupportFragmentManager()));
  }
}
```

(from ViewPager/Columns1/app/src/main/java/com/commonsware/android/pagercolumns/MainActivity.java)

Of course, we skip the `FragmentTransaction` if the fragments already exist, such as due to a screen rotation configuration change.

The Results

On a phone, the `ViewPager`-based layout looks pretty much as it did before:

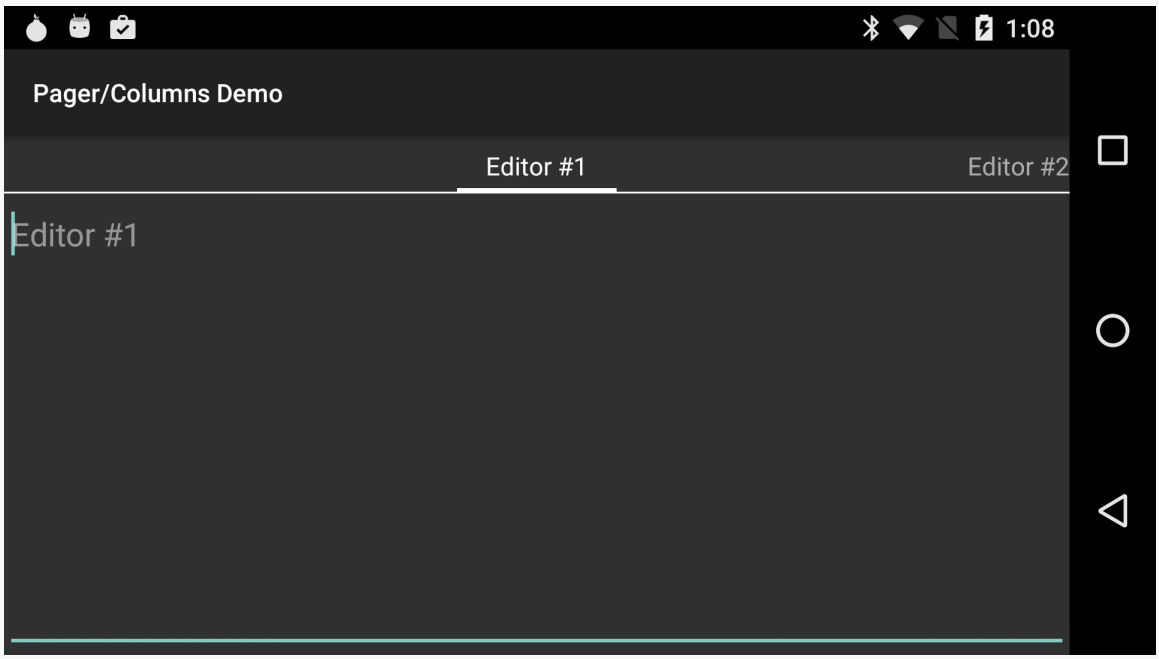


Figure 659: A ViewPager. Again.

However, on a tablet, we get our three editors side-by-side:

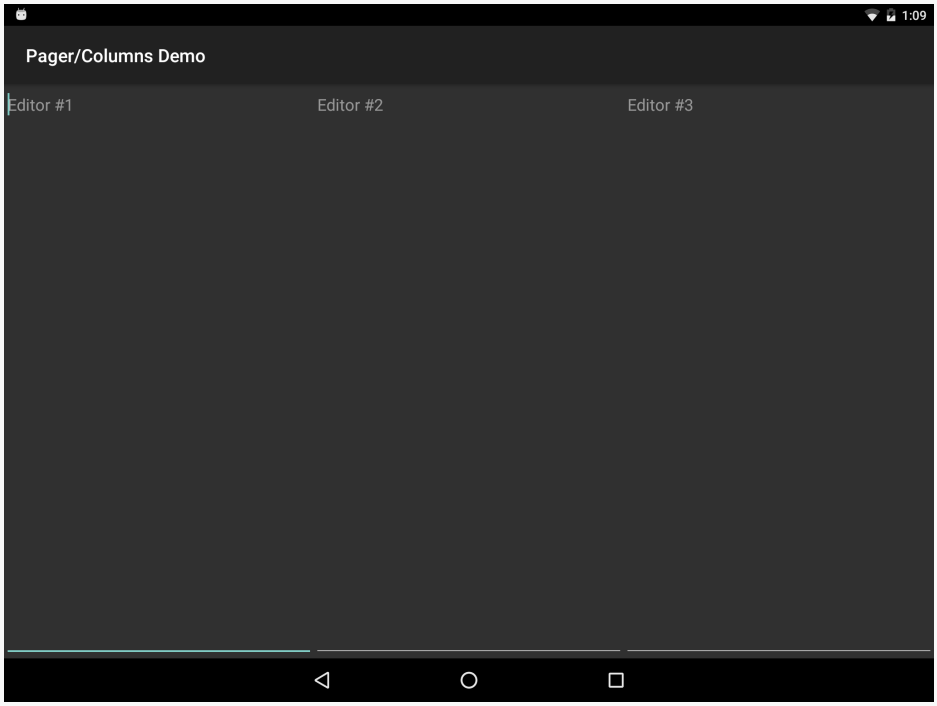


Figure 660: Same App, Large-Screen Layout with Side-By-Side Editors

The Limitations

The simplified large-screen layout does not contain any indicators above the three editors. This could be added by simple changes to the `res/layout-large/main.xml` layout resource, if desired.

The bigger limitation is that this only works if you want the same look in all configurations *except* screen size, and if the screen size never changes. However, it is eminently possible that you will want to have a different mix than that, such as using the three-column approach only on large-screen *landscape* layouts, using `ViewPager` everywhere else. In that case, our approach breaks down, as we will have different fragments inside the pager and outside the pager, meaning that we will lose our data on a configuration change. Addressing this issue is covered in the next two sections.

Focus Management and Accessibility

As developers, we are very used to creating apps that are designed to be navigated by touch, with users tapping on widgets and related windows to supply input.

However, not all Android devices have touchscreens, and not all Android users use touchscreens.

Internationalization (i18n) and localization (L10n) give you opportunities to expand your user base to audiences beyond your initial set, based on language. Similarly, you can expand your user base by offering support for non-touchscreen input and output. Long-term, the largest user base of these features may be those with televisions augmented by Android, whether via Android TV, OUYA consoles, or whatever. Short-term, the largest user base of these features may be those for whom touchscreens are rarely a great option, such as the blind. Supporting those with unusual requirements for input and output is called *accessibility* (a11y), and represents a powerful way for you to help your app distinguish itself from competitors.

In this chapter, we will first examine how to better handle focus management, and then segue into examining what else, beyond supporting keyboard-based input, can be done in the area of accessibility.

Prerequisites

Understanding this chapter requires that you have read the core chapters and are familiar with the concept of widgets having focus for user input.

Prepping for Testing

To test focus management, you will need an environment that supports "arrow key" navigation. Here, "arrow key" also includes things like D-pads or trackballs – basically, anything that navigates by key events instead of by touch events.

Examples include:

- The Android emulator, with the `DPad support` hardware property set to `yes`
- Phones that have actual D-pads, trackballs, arrow keys, or the like
- Television-based Android environments, such as Android TV or the OUYA console
- Devices that have dedicated keyboard accessories, such as the keyboard "slice" available for the ASUS Transfomer series of tablets
- A standard Android device accessed via a Bluetooth keyboard, gamepad, or similar sort of pointing device

Hence, even if the emulator will be insufficient for your needs, you should be able to set up a hardware test environment relatively inexpensively. Most modern Android devices support Bluetooth keyboards, and such keyboards frequently can be obtained at low relative cost.

For accessibility beyond merely focus control, you will certainly want to enable TalkBack, via the Accessibility area of the Settings app. This will cause Android to verbally announce what is on the screen, by means of its text-to-speech engine.

On Android 4.0 and higher devices, enabling Talkback will also optionally enable "Explore by Touch". This allows users to tap on items (e.g., icons in a `GridView`) to have them read aloud via TalkBack, with a double-tap to actually perform what ordinarily would require a single-tap without "Explore by Touch".

Controlling the Focus

Android tries its best to have intelligent focus management "out of the box", without developer involvement. Many times, what it offers is sufficient for your needs. Other times, though, the decisions Android makes are inappropriate:

- Trying to navigate in a certain direction (e.g., right) moves focus to a widget that is not logically what should have the focus

- Focus has other side effects, like showing the soft keyboard on an `EditText` widget, that is not desirable

Hence, if you feel that you need to take more control over how focus management is handled, you have many means of doing so, covered in this section.

Establishing Focus

In order for a widget to get the focus, it has to be focusable.

You might think that the above sentence was just a chance for the author to be witty. It was... a bit. But there are actually two types of "focusable" when it comes to Android apps:

- Is it focusable when somebody is using a pointing device or the keyboard?
- Is it focusable in touch mode?

There are three major patterns for the default state of a widget:

1. Some are initially focusable in both cases (e.g., `EditText`)
2. Some are focusable in non-touch mode but are not focusable in touch mode (e.g., `Button`)
3. Some are not focusable in either mode (e.g., `TextView`)

So, when a `Button` is not focusable in touch mode, that means that while the button will take the focus when the user navigates to it (e.g., via keys), the button will *not* take the focus when the user simply taps on it.

You can control the focus semantics of a given widget in four ways:

- You can use `android:focusable` and `android:focusableInTouchMode` in a layout
- You can use `setFocusable()` and `setFocusableInTouchMode()` in Java

We will see examples of these shortly.

Requesting (or Abandoning) Focus

By default, the focus will be granted to the first focusable widget in the activity, starting from the upper left. Often times, this is a fine solution.

If you want to have some other widget get the focus (assuming that the widget is focusable, per the section above), you have two choices:

1. Call `requestFocus()` on the widget in question
2. You can give the widget's layout element a child element, named `<requestFocus />`, to stipulate that this widget should be the one to get the focus

Note that this is a child element, not an attribute, as you might ordinarily expect.

For example, let's look at the `Focus/Sampler` sample project, which we will use to illustrate various focus-related topics.

Our main activity, creatively named `MainActivity`, loads a layout named `request_focus.xml`, and demonstrates the `<requestFocus />` element:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/a_button"/>

  <EditText
    android:id="@+id/editText1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/first_field"
    android:hint="@string/str_1st_field"
    android:inputType="text"/>

  <EditText
    android:id="@+id/editText2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/second_field"
    android:hint="@string/str_2nd_field"
    android:inputType="text">

    <requestFocus/>
```

```
  </EditText>

</LinearLayout>
```

(from Focus/Sampler/app/src/main/res/layout/request_focus.xml)

Here, we have three widgets in a horizontal `LinearLayout`: a `Button`, and two `EditText` widgets. The second `EditText` widget has the `<requestFocus />` child element, and so it gets the focus when we display our launcher activity:

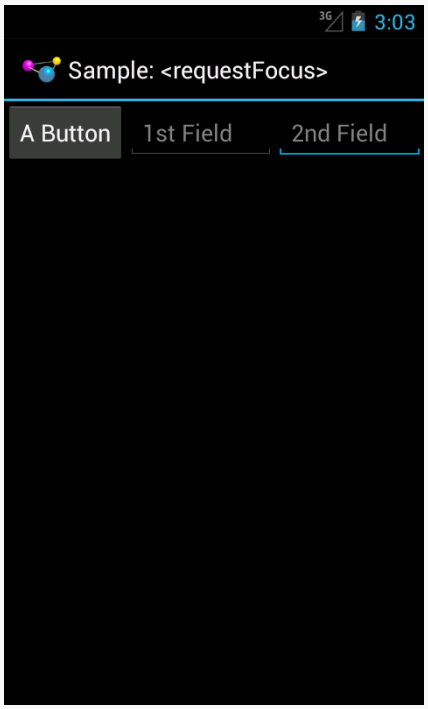


Figure 661: Focus Sampler, Showing Requested Focus

If we had skipped the `<requestFocus />` element, the focus would have wound up on the first `EditText`... assuming that we are working in touch mode. If the activity had been launched via the pointing device or keyboard, then the `Button` would have the focus, because the `Button` is focusable in non-touch mode by default.

Calling `requestFocus()` from Java code gets a bit trickier. There are a few flavors of the `requestFocus()` method on `View`, of which two will be the most popular:

- An ordinary zero-argument `requestFocus()`
- A one-argument `requestFocus()`, with the argument being the direction in which the focus should theoretically be coming from

You might look at the description of the second flavor and decide that the zero-argument `requestFocus()` looks a lot easier. And, sometimes it will work. However, sometimes it will not, as is the case with our second activity, `RequestFocusActivity`.

In this activity, our layout (`focusable_button`) is a bit different:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <EditText
    android:id="@+id/editText1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/first_field"
    android:hint="@string/str_1st_field"
    android:inputType="text"/>

  <EditText
    android:id="@+id/editText2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/second_field"
    android:hint="@string/str_2nd_field"
    android:inputType="text">
  </EditText>

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:focusableInTouchMode="true"
    android:text="@string/a_button"/>

</LinearLayout>
```

(from Focus/Sampler/app/src/main/res/layout/focusable_button.xml)

Here, we put the `Button` last instead of first. We have no `<requestFocus />` element anywhere, which would put the default focus on the first `EditText` widget. And, our `Button` has `android:focusableInTouchMode="true"`, so it will be focusable regardless of whether we are in touch mode or not.

In `onCreate()` of our activity, we use the one-parameter version of `requestFocus()` to give the `Button` the focus:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.focusable_button);
  initActionBar();

  button=findViewById(R.id.button1);
  button.requestFocus(View.FOCUS_RIGHT);
  button.setOnClickListener(this);
}
```

(from Focus/Sampler/app/src/main/java/com/commonsware/android/focus/RequestFocusActivity.java)

If there were only the one `EditText` before the `Button`, the zero-argument `requestFocus()` works. However, with a widget between the default focus and our `Button`, the zero-argument `requestFocus()` does not work, but using `requestFocus(View.FOCUS_RIGHT)` does. This tells Android that we want the focus, and it should be as if the user is moving to the right from where the focus currently lies.

All of our activities inherit from a `BaseActivity` that manages our action bar, with an overflow menu to get to the samples and the app icon to get to the original activity.

So, if you run the app and choose “Request Focus” from the overflow menu, you will see:

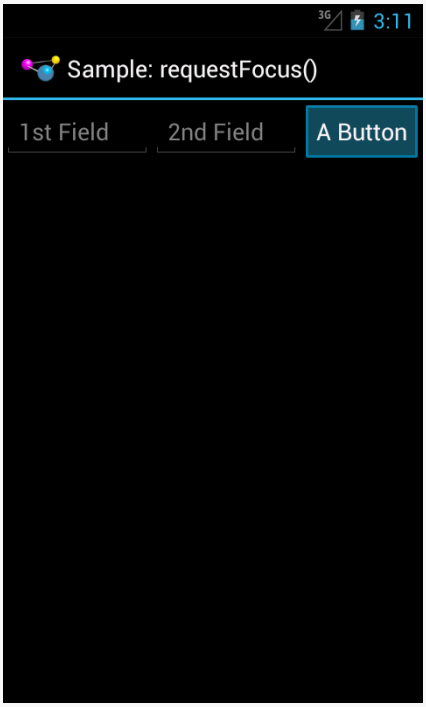


Figure 662: Focus Sampler, Showing Manually-Requested Focus

We also wire up the `Button` to the activity for click events, and in `onClick()`, we call `clearFocus()` to abandon the focus:

```
@Override
public void onClick(View v) {
  button.clearFocus();
}
```

(from Focus/Sampler/app/src/main/java/com/commonsware/android/focus/RequestFocusActivity.java)

What `clearFocus()` will do is return to the original default focus for this activity, in our case the first `EditText`:

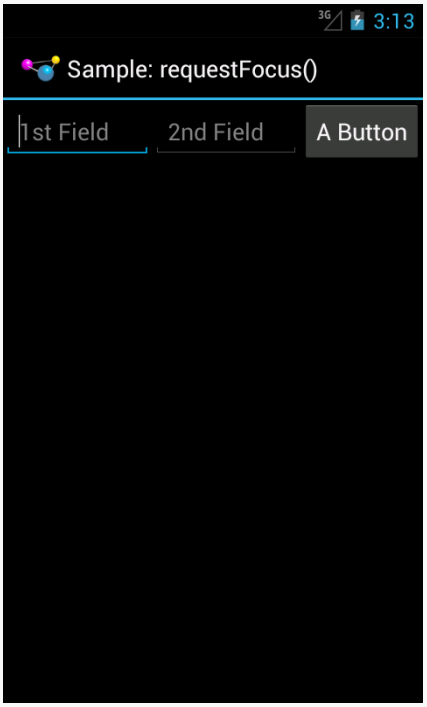


Figure 663: Focus Sampler, After Clearing the Focus

Focus Ordering

Beyond manually placing the focus on a widget (or manually clearing that focus), you can also override the focus order that Android determines automatically. While Android's decisions usually are OK, they may not be optimal.

A widget can use `android:nextFocus...` attributes in the layout file to indicate the widget that should get control on a focus change in the direction indicated by the `...` part. So, `android:nextFocusDown`, applied to Widget A, indicates which widget should receive the focus if, when the focus is on Widget A, the user "moves down" (e.g., presses a DOWN key, presses the down direction on a D-pad). The same logic holds true for the other three directions (`android:nextFocusLeft`, `android:nextFocusRight`, and `android:nextFocusUp`).

For example, the `res/layout/table.xml` resource in the `FocusSampler` project is based on the `TableLayout` sample from early in this book, with a bit more focus control:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:stretchColumns="1">

  <TableRow>

    <TextView android:text="@string/url"/>

    <EditText
      android:id="@+id/entry"
      android:layout_span="3"
      android:inputType="text"
      android:nextFocusRight="@+id/ok"/>
  </TableRow>

  <TableRow>

    <Button
      android:id="@+id/cancel"
      android:layout_column="2"
      android:text="@string/cancel"/>

    <Button
      android:id="@+id/ok"
      android:text="@string/ok"/>
  </TableRow>

</TableLayout>
```

(from Focus/Sampler/app/src/main/res/layout/table.xml)

In the original `TableLayout` sample, by default, pressing either RIGHT or DOWN while the `EditText` has the focus will move the focus to the “Cancel” button. This certainly works. However, it does mean that there is no single-key means of moving from the `EditText` to the “OK” button, and it would be nice to offer that, so those using the pointing device or keyboard can quickly move to either button.

This is a matter of overriding the default focus-change behavior of the `EditText` widget. In our case, we use `android:nextFocusRight="@+id/ok"` to indicate that the “OK” button should get the focus if the user presses RIGHT from the `EditText`. This gives RIGHT and DOWN different behavior, to reach both buttons.

Scrolling and Focusing Do Not Mix

Let's suppose that you have a UI design with a fixed bar of widgets at the top (e.g., action bar), a `ListView` dominating the activity, and a panel of widgets at the bottom (e.g., a `Toolbar`). This is a common UI pattern on iOS, though it is relatively uncommon on Android nowadays. You used to see it with the so-called "split action bar", which is now officially deprecated as a pattern:

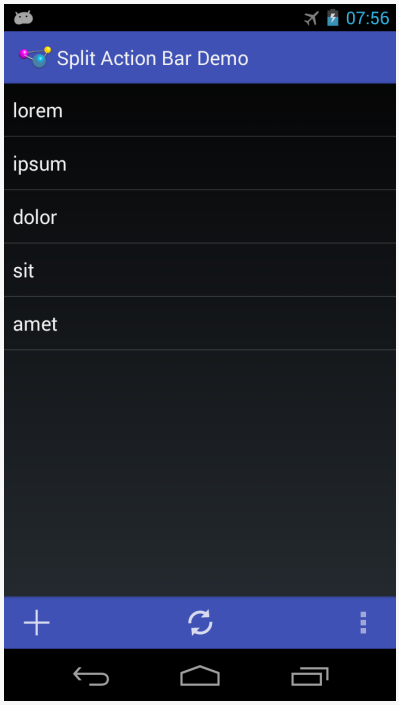


Figure 664: Split Action Bar

However, this UI pattern does not work well for those using pointing devices or keyboards for navigation. In order to get to the bottom panel of widgets, they will have to scroll through the entire list first, because scrolling trumps focus changes. So while this is easy to navigate via a touchscreen, it is a major problem to navigate for those not using a touchscreen.

Similarly, if the user has scrolled down the list, and now wishes to get to the action bar at the top, the user would have to scroll all the way to the top of the list first.

Workarounds include:

- Overriding focus control such that left and right navigation from the list moves you to the action bar or bottom `Toolbar` (e.g., left moves you to the action bar, right moves you to the `Toolbar`)
- In a television setup, having the "action bar" be vertical down the left, and the tools be vertical down the right, so you automatically get the left/right navigation to move between these "zones"
- Eliminating the `Toolbar` entirely, moving those items instead to the action bar, or perhaps an action mode (a.k.a., contextual action bar) if the items are only relevant if the user checks one or more items in the list
- Offer a hotkey, separate from navigation, that repositions the focus (e.g., Ctrl-A to jump to the action bar), if you believe that users will read your documentation to discover this key combination

Accessibility and Focus

People suffering from impaired vision, including the blind, have had to rely heavily on proper keyboard navigation for their use of Android apps, at least prior to Android 4.0 and "Explore by Touch". These users need focus to be sensible, so that they can find their way through your app, with TalkBack supplying prompts for what has the focus. Having widgets that are unreachable in practice will eliminate features from your app for this audience, simply because they cannot get to them.

"Explore by Touch" provides accessibility assistance without reliance upon proper focus. However:

- "Explore by Touch" is new to Android 4.0, and a few visually-impaired users will be using older devices
- "Explore by Touch" is less reliable than keyboard-based navigation, insofar as users have to remember specific screen locations (and get to them without seeing those locations), rather than simply memorizing certain key combinations
- "Explore by Touch", by requiring additional taps (e.g., double-tap to tap a `Button`), may cause some challenges when the UI itself requires additional taps (e.g., a double-tap on a widget to perform an action — is this now a triple-tap in "Explore by Touch" mode?)
- "Explore by Touch" is mostly for the visually impaired, and does not help others that might benefit from key-based navigation (e.g., people with limited motor control)

So, even though "Explore by Touch" will help people use apps that cannot be navigated purely through key events, the better you can support keyboards, the better off your users will be.

Accessibility Beyond Focus

While getting focus management correct goes a long way towards making your application easier to use, it is not the only thing to consider for making your application truly accessible by all possible users. This section covers a number of other things that you should consider as part of your accessibility initiatives.

Content Descriptions

For TalkBack to work, it needs to have something useful to read aloud to the user. By default, for most widgets, all it can say is the type of widget that has the focus (e.g., "a checkbox"). That does not help the TalkBack-reliant user very much.

Please consider adding `android:contentDescription` attributes to most of your widgets, pointing to a string resource that briefly describes the widget (e.g., "the Enabled checkbox"). This will be used in place of the basic type of widget by TalkBack.

Classes that inherit from `TextView` will use the text caption of the widget by default, so your `Button` widgets may not need `android:contentDescription` if their captions will make sense to TalkBack users.

However, with an `EditText`, since the text will be what the user types in, the text is not indicative of the widget itself. Android will first use your `android:hint` value, if available, falling back to `android:contentDescription` if `android:hint` is not supplied.

Also, bear in mind that if the widget changes purpose, you need to change your `android:contentDescription` to match. For example, suppose you have a media player app with an `ImageButton` that you toggle between "play" and "pause" modes by changing its image. When you change the image, you also need to change the `android:contentDescription` as well, lest sighted users think the button will now "pause" while blind users think that the button will now "play".

Labels

Sometimes, we have `TextView` widgets that serve as on-screen labels for some adjacent other widget, such as an `EditText`. In those cases, you can associate the two by using `android:labelFor` on the `TextView`, supplying the ID of the widget for which the `TextView` is a label. This will help accessibility tools properly announce the widgets, as otherwise those tools do not know that these widgets are related.

Custom Widgets and Accessibility Events

The engine behind TalkBack is an accessibility service. Android ships with some, like TalkBack, and third parties can create other such services.

Stock Android widgets generate relevant accessibility events to feed data into these accessibility services. That is how `android:contentDescription` gets used, for example — on a focus change, stock Android widgets will announce the widget that just received the focus.

If you are creating custom widgets, you may need to raise your own accessibility events. This is particularly true for custom widgets that draw to the `Canvas` and process raw touch events (rather than custom widgets that merely aggregate existing widgets).

The Android developer documentation provides instructions for when and how to supply these sorts of events.

Announcing Events

Sometimes, your app will change something about its visual state in ways that do not get picked up very well by any traditional accessibility events. For example, you might use `GestureDetector` to handle some defined library of gestures and change state in your app. Those state changes may have visual impacts, but `GestureDetector` will not know what those are and therefore cannot supply any sort of accessibility event about them.

To help with this, API Level 16 added `announceForAccessibility()` as a method on `View`. Just pass it a string and that will be sent out as an “announcement” style of `AccessibilityEvent`. Your code leveraging `GestureDetector`, for example, could use this to explain the results of having applied the gesture.

Font Selection and Size

For users with limited vision, being able to change the font size is a big benefit. Android 4.0 finally allows this, via the Settings app, so users can choose between small, normal, large, and huge font sizes. Any place where text is rendered and is measured in `sp` will adapt.

The key, of course, is the `sp` part.

`sp` is perhaps the most confusing of the available dimension units in Android. `px` is obvious, and `dp` (or `dip`) is understandable once you recognize the impacts of screen density. Similarly, `in`, `mm`, and `pt` are fairly simple, at least once you remember that `pt` is 1/72nd of an inch.

If the user has the font scale set to "normal", `sp` equates to `dp`, so a dimension of `30sp` and `30dp` will be the same size. However, values in `dp` do not change based on font scale; values in `sp` will increase or decrease in physical size based upon the user's changes to the font scale.

We can see how this works in the `Accessibility/FontScale` sample project.

In our layout (`res/layout/activity_main.xml`), we have six pieces of text: two each (regular and bold) measured at `30px`, `30dp`, and `30sp`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/LinearLayout1"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="@string/normal_30px"
    android:textSize="30px"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bold_30px"
    android:textSize="30px"
```

```
    android:textStyle="bold"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="@string/normal_30dp"
    android:textSize="30dp"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bold_30dp"
    android:textSize="30dp"
    android:textStyle="bold"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="@string/normal_30sp"
    android:textSize="30sp"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bold_30sp"
    android:textSize="30sp"
    android:textStyle="bold"
    tools:context=".MainActivity"/>

</LinearLayout>
```

(from Accessibility/FontScale/app/src/main/res/layout/activity_main.xml)

You will be able to see the differences between `30px` and `30dp` on any Android OS release, simply by running the app on devices with different densities. To see the changes between `30dp` and `30sp`, you will need to run the app on an Android 4.0+ device or emulator and change the font scale from the Settings app (typically in the Display section).

Here is what the text looks like with a normal font scale:

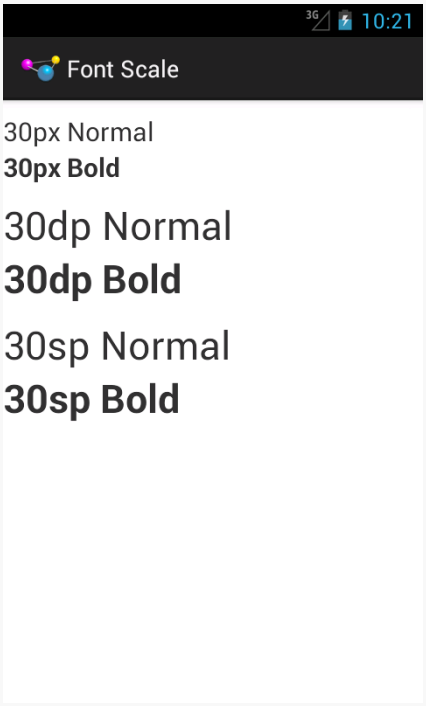


Figure 665: Fonts at Normal Scale

As you can see, `30dp` and `30sp` are equivalent.

If we raise the font scale to "large", the `30sp` text grows to match:

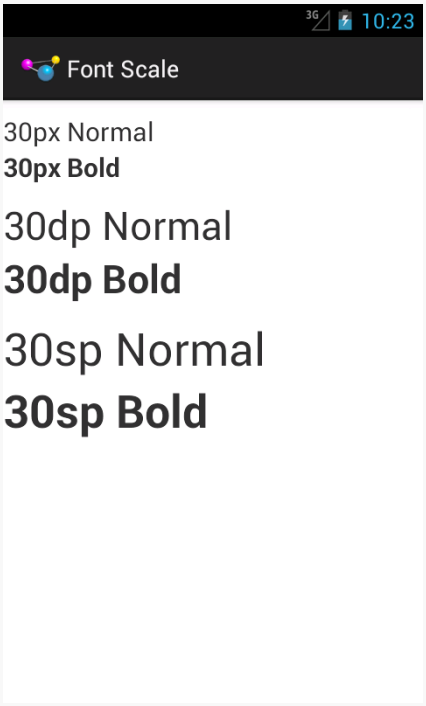


Figure 666: Fonts at Large Scale

Moving to "huge" scale increases the `30sp` text size further:

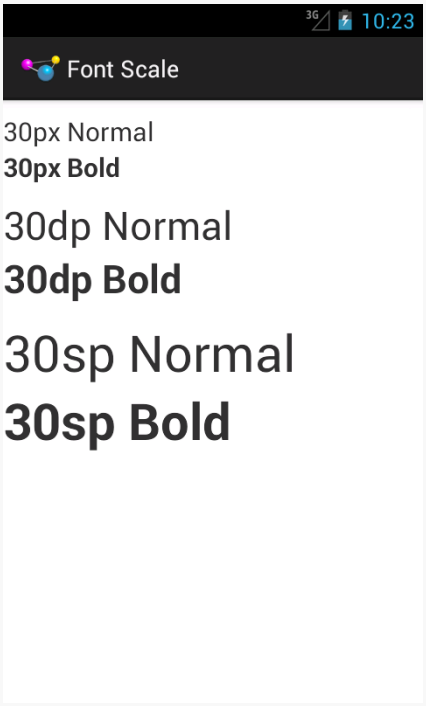


Figure 667: Fonts at Huge Scale

In the other direction, some users may elect to drop their font size to "small", with a corresponding impact on the `30sp` text:

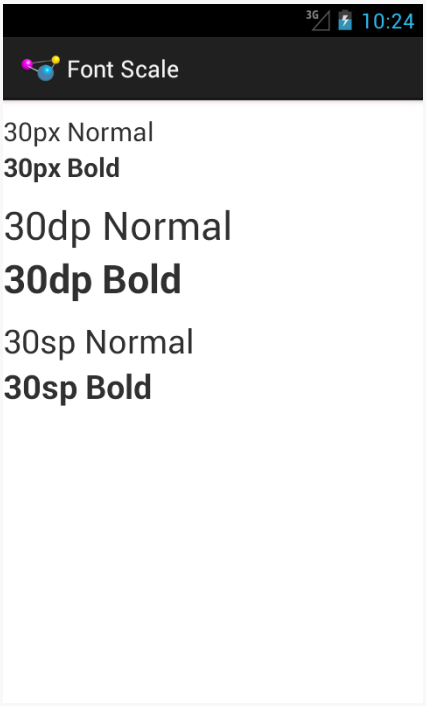


Figure 668: Fonts at Small Scale

As a developer, your initial reaction may be to run away from `sp`, because you do not control it. However, just as Web developers should deal with changing font scale in Web browsers, Android developers should deal with changing font scale in Android apps. Remember: the user is changing the font scale because the *user* feels that the revised scale is easier for them to use. Blocking such changes in your app, by avoiding `sp`, will not be met with love and adoration from your user base.

Also, bear in mind that changes to the font scale represent a configuration change. If your app is in memory at the time the user goes into Settings and changes the scale, if the user returns to your app, each activity that comes to the foreground will undergo the configuration change, just as if the user had rotated the screen or put the device into a car dock or something.

Widget Size

Users with ordinary sight already have trouble with tiny widgets, as they are difficult to tap upon.

Users trying to use the Explore by Touch facility added in Android 4.1 have it worse, as they cannot even see (or see well) the tiny target you are expecting them to tap

upon. They need to be able to reliably find your widget based on its relative position on the screen, and their ability to do so will be tied, in part, on widget size.

The Android design guidelines recommend 7-10mm per side minimum sizes for tappable widgets. In particular, they recommend 48dp per side, which results in a size of about 9mm per side.

You also need to consider how closely packed your widgets are. The closer the tap targets lie, the more likely it is that all users — whether using Explore by Touch or not — will accidentally tap on the wrong thing. Google recommends 8dp or more of margin between widgets. Also note that the key is *margins*, as while increasing padding might visually separate the widgets, the padding is included as part of the widget from the standpoint of touch events. While padding may help users with ordinary sight, margins provide similar help while also being of better benefit to those using Explore by Touch.

Gestures and Taps

If you employ gestures, be careful when employing the same gesture in different spots for different roles, particularly within the same activity.

For example, you might use a horizontal swipe to the right to switch pages in a `ViewPager` in some places and remove items from a `ListView` in others. While there may be visual cues to help explain this to users with ordinary sight, it may be far less obvious what is going on for TalkBack users. This is even more true if you are somehow combining these things (e.g., the `ListView` in question is in a page of the `ViewPager`).

Also, be a bit careful as you "go outside the box" for tap events. You might decide that a double-tap, or a two-finger tap, has special meaning on some widgets. Make sure that this still works when users use Explore by Touch, considering that the first tap will be "consumed" by Explore by Touch to announce the widget being tapped upon.

Enhanced Keyboard Support

All else being equal, users seeking accessibility assistance will tend to use keyboards when available. For users with limited (or no) sight, tactile keyboards are simply easier to use than touchscreens. For users with limited motor control, external devices that interface as keyboards may allow them to use devices that otherwise they could not.

Of course, plenty of users will use keyboards outside of accessibility as well. For example, devices like the ASUS Transformer series form perfectly good "netbook"-style devices when paired with their keyboards.

Hence, consider adding hotkey support, to assist in the navigation of your app. Some hotkeys may be automatically handled (e.g., Ctrl-C for copy in an `EditText`). However, in other cases you may wish to add those yourself (e.g., Ctrl-C for "copy" with respect to a checklist and its selected rows, in addition to a "copy" action mode item).

API Level 11 adds `KeyEvent` support for methods like `isCtrlPressed()` to detect meta keys used in combination with regular keys.

Audio and Haptics

Of course, another way to make your app more accessible is to provide alternative modes of input and output, beyond the visual.

Audio is popular in this regard:

- Using tones or clicks to reinforce input choices
- Integrating your own text-to-speech to augment TalkBack
- Integrating speech recognition for simple commands

However, bear in mind that deaf users will be unable to hear your audio. You are better served using both auditory and visual output, not just one or the other.

In some cases, haptics can be helpful for input feedback, by using the `Vibrator` system service to power the vibration motor. While most users will be able to feel vibrations, the limitation here is whether the device is capable of vibrating:

- Some tablets lack a vibration motor
- Television-based Android environment may or may not have some sort of vibration output (e.g., remote controls probably will not, but game controllers might)
- Devices not held in one's hand, such as those in a dock, will make haptics less noticeable

So, audio and vibration can help augment visual input and output, though they should not be considered complete replacements except in rare occurrences.

Color and Color Blindness

Approximately 8% of men (and 0.5% of women) in the world are colorblind, meaning that they cannot distinguish certain close colors:

> ...It's not that colorblind people (in most cases) are incapable or perceiving "green," instead they merely distinguish fewer shades of green than you do. So where you see three similar shades of green, a colorblind user might only see one shade of green.

(from "Tips for Designing for Colorblind Users")

Hence, relying solely on colors to distinguish different items, particularly when required for user input, is not a wise move.

Make sure that there is something more to distinguish two pieces of your UI than purely a shift in color, such as:

- Labels or icons
- Textures (e.g., solid vs. striped)
- Borders (e.g., drop shadow)

Accessibility Beyond Impairment

Accessibility is often tied to impaired users: ones with limited (or no) sight, ones with limited (or no) hearing, ones with limited motor control, etc.

In reality, accessibility is for *situations* where users may have limitations. For example, a user who might not normally think of himself as "impaired" has limited sight, hearing, and motor control when those facilities are already in use, such as while driving.

Hence, offering features that help with accessibility can benefit *all* your users, not just ones you think of as "impaired". For example:

- Offer a UI mode with an eye towards use in low-visibility situations that can either be manually invoked (e.g., via a preference) or automatically invoked (e.g., via a car dock)
- Offer voice input (commands) and output (text-to-speech) — iOS's Siri is not just for the blind, after all

- Offer hotkeys, not only to help those requiring a keyboard as their primary mode of input (e.g., blind users minimizing touchscreen use), but to help those who opt into using it for input (e.g., using a keyboard with an Android tablet in lieu of a traditional notebook or netbook)

Miscellaneous UI Tricks

While well-written GUI frameworks are better organized than XKCD's take on home organization, there are always a handful of tidbits that do, indeed, get categorized as "miscellaneous".

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having an appreciation for XKCD is welcome, but optional.

Full-Screen and Lights-Out Modes

Full-screen mode, in Android parlance, means removing any system-supplied "bars" from the screen: title bar, action bar, status bar, system bar, navigation bar, etc. You might use this for games, video players, digital book readers, or other places where the time spent in an activity is great enough to warrant removing some of the normal accouterments to free up space for whatever the activity itself is doing.

Lights-out mode, in Android parlance, is where you take the system bar or navigation bar and dim the widgets inside of them, such that the bar is still usable, but is less visually distracting. This is a new concept added in Android 3.0 and has no direct analogue in Android 1.x or 2.x.

Android 1.x/2.x

To have an activity be in full-screen mode, you have two choices:

1. Having the activity use a theme of `Theme.NoTitleBar.Fullscreen` (or some custom theme that inherits from `Theme.NoTitleBar.Fullscreen`)
2. Execute the following statements in `onCreate()` of your activity *before* calling `setContentView()`:

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                     WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

The first statement removes the title bar or action bar. The second statement indicates that you want the activity to run in full-screen mode, hiding the status bar.

Android 4.0+

Things got significantly more messy once we started adding in the system bar (and, later, the navigation bar as the replacement for the system bar). Since these bars provide the user access to HOME, BACK, etc., it is usually important for them to be available. Android's behavior, therefore, varies in how you ask for something to happen and what then happens, based upon whether the device is a phone or a tablet.

The Activities/FullScreen sample project tries to enumerate some of the possibilities. On an Android 4.0 device, we have three `RadioButtons`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <RadioGroup
    android:id="@+id/screenStyle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <RadioButton
      android:id="@+id/normal"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:checked="true"
      android:text="@string/display_normal"/>

    <RadioButton
      android:id="@+id/lowProfile"
      android:layout_width="wrap_content"
```

```
      android:layout_height="wrap_content"
      android:text="@string/display_low_profile"/>

    <RadioButton
      android:id="@+id/hideNav"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_hide_navigation"/>
  </RadioGroup>

  <Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:text="@string/something_at_the_bottom"/>

</RelativeLayout>
```

(from Activities/FullScreen/app/src/main/res/layout/main.xml)

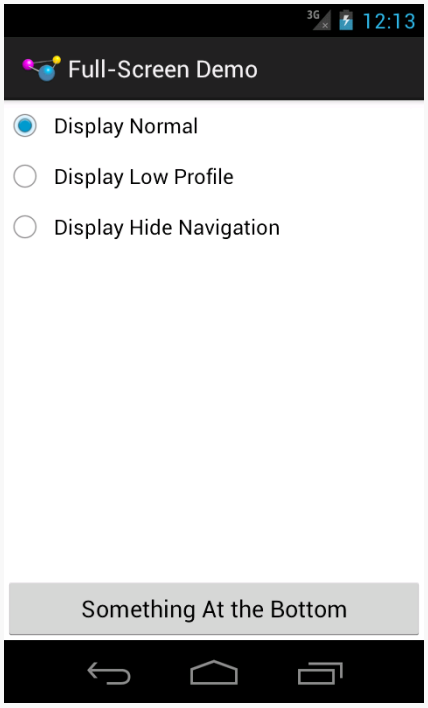


Figure 669: Sample UI, As Initially Launched on Android 4.0

...while on Android 4.1 or higher, we have another two possibilities:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <RadioGroup
    android:id="@+id/screenStyle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <RadioButton
      android:id="@+id/normal"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:checked="true"
      android:text="@string/display_normal"/>

    <RadioButton
      android:id="@+id/lowProfile"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_low_profile"/>

    <RadioButton
      android:id="@+id/hideNav"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_hide_navigation"/>

    <RadioButton
      android:id="@+id/hideStatusBar"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/hide_status_bar"/>

    <RadioButton
      android:id="@+id/fullScreen"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_full_screen"/>
  </RadioGroup>

  <Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:text="@string/something_at_the_bottom"/>
```

```
</RelativeLayout>
```

(from Activities/FullScreen/app/src/main/res/layout-v16/main.xml)

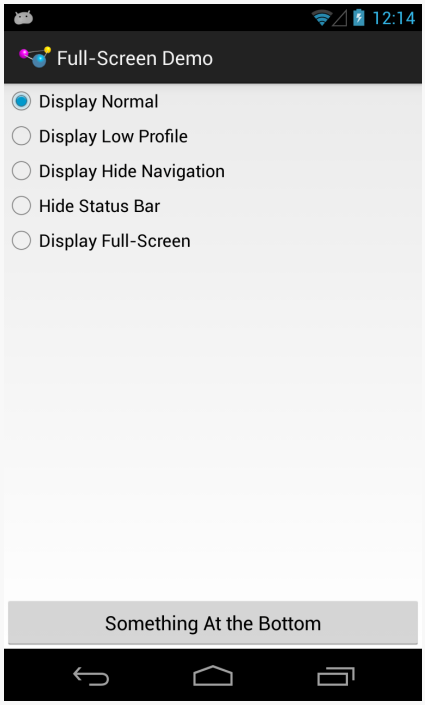


Figure 670: Sample UI, As Initially Launched on a Nexus 4/Android 4.2

Controlling the full-screen and lights-out modes is managed via a call to `setSystemUiVisibility()`, a method on `View`. You pass in a value made up of an OR'd (`|`) set of flags indicating what you want the visibility to be, with the default being normal operation. Hence, in the screenshot above, you see a Nexus 4 in normal mode. Here is the same UI on a Nexus 10 in normal mode:

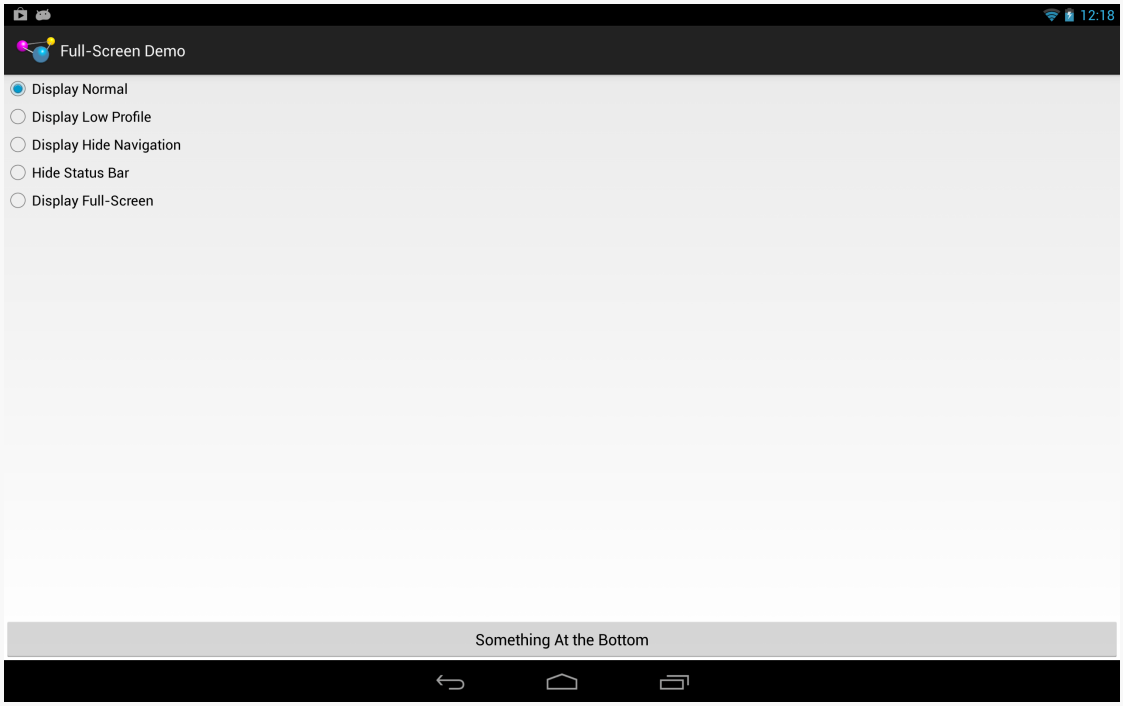


Figure 671: Sample UI, As Initially Launched on a Nexus 10/Android 4.2

Lights-out, or low-profile mode, is achieved by calling `setSystemUiVisibility()` with the `View.SYSTEM_UI_FLAG_LOW_PROFILE` flag. This will dim the navigation or system bar, so the bar is there and the buttons are still active, but that they are less visually intrusive:

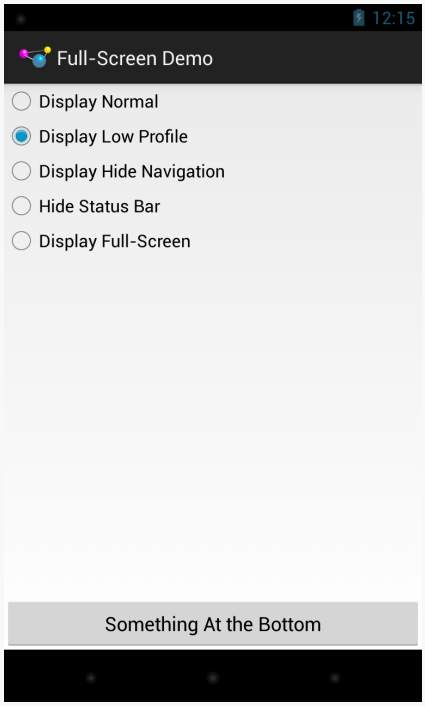


Figure 672: Sample UI, Lights-Out Mode, Nexus 4/Android 4.2

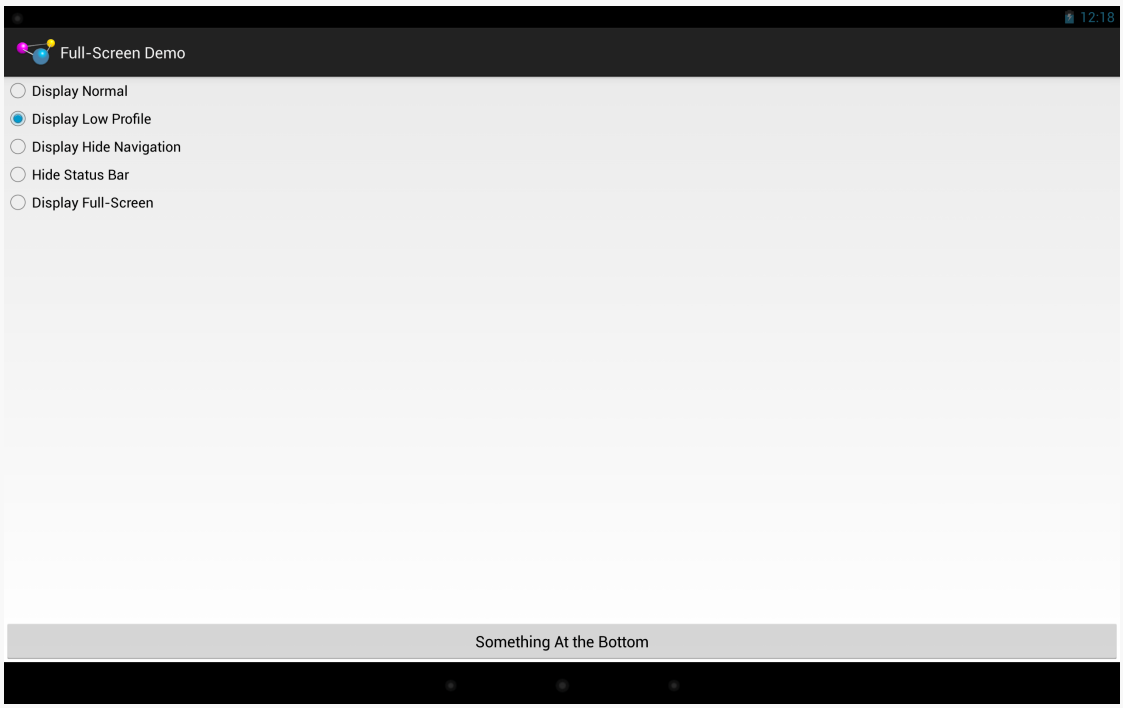


Figure 673: Sample UI, Lights-Out Mode, Nexus 10/Android 4.2

You can temporarily hide the navigation bar (or system bar) by passing `View.SYSTEM_UI_FLAG_HIDE_NAVIGATION` to `setSystemUiVisibility()`. The bar will disappear, until the user touches the UI, in which case the bar reappears:

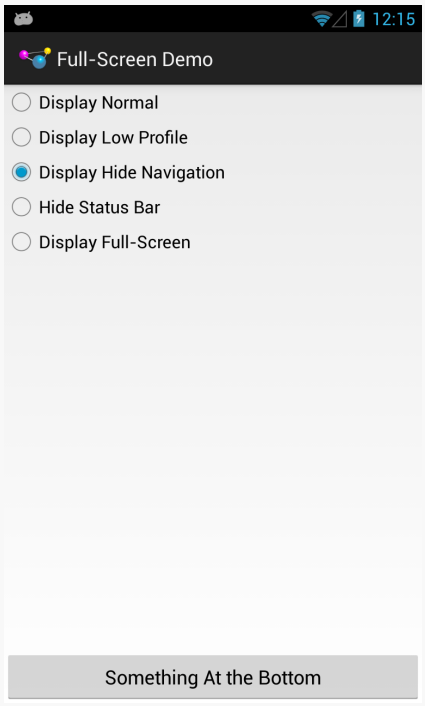


Figure 674: Sample UI, Hidden-Navigation Mode, Nexus 4/Android 4.2

Figure 675: Sample UI, Hidden-Navigation Mode, Nexus 10/Android 4.2

Similarly, you can hide the status bar by passing `View.SYSTEM_UI_FLAG_FULLSCREEN` to `setSystemUiVisibility()`. However, despite this flag's name, it does not affect the navigation or system bar:

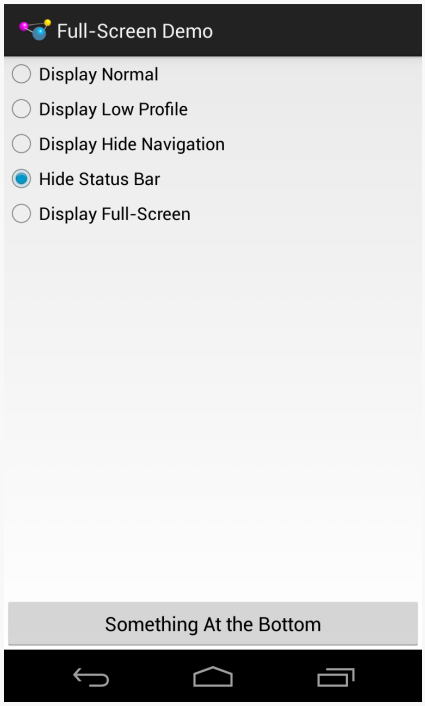


Figure 676: Sample UI, "Full-Screen" Mode, Nexus 4/Android 4.2

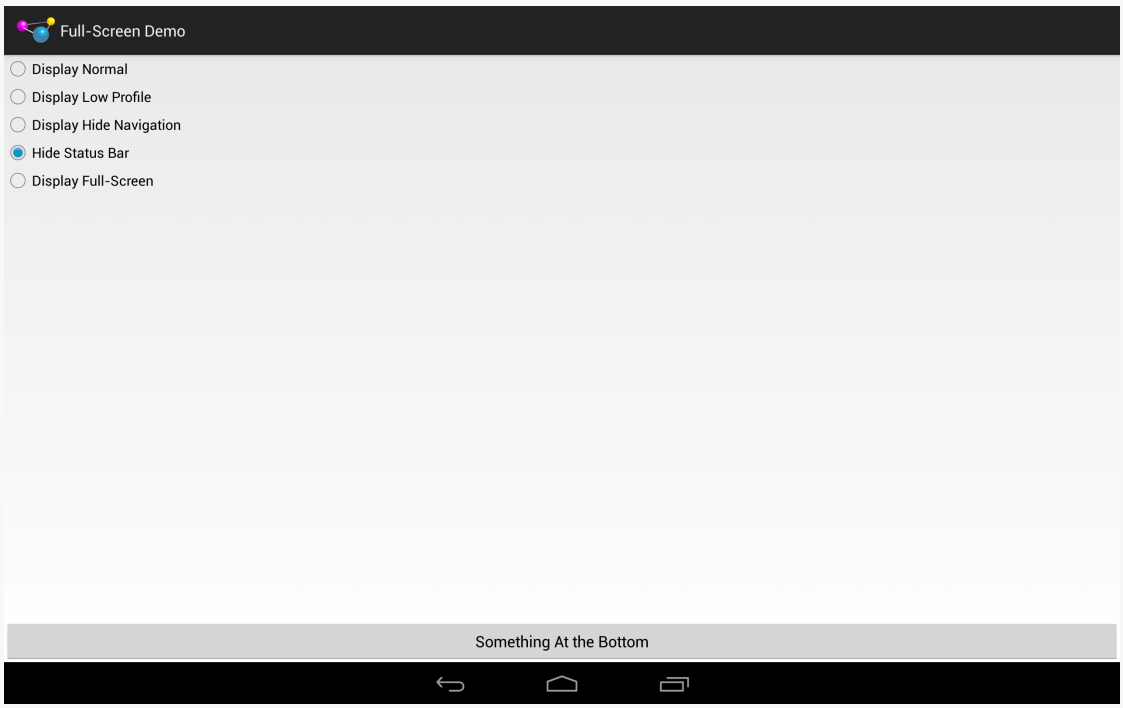


Figure 677: Sample UI, "Full-Screen" Mode, Nexus 10/Android 4.2

Hence, to hide both the status bar and the navigation or system bar, you need to pass both flags (`View.SYSTEM_UI_FLAG_FULLSCREEN | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION`):

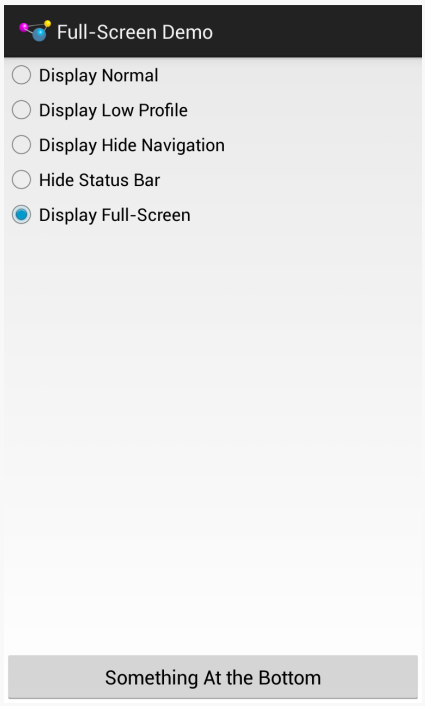


Figure 678: Sample UI, True Full-Screen Mode, Nexus 4/Android 4.2

Figure 679: Sample UI, True Full-Screen Mode, Nexus 10/Android 4.2

Note that showing and hiding the `ActionBar` is also possible, via calls to `show()` and `hide()`, respectively.

Offering a Delayed Timeout

Android makes it easy for activities to keep the screen on while the activity is in the foreground, by means of `android:keepScreenOn` and `setKeepScreenOn()`.

However, these are very blunt instruments, and too many developers simply ask to keep the screen on constantly, even when that is not needed and can cause excessive battery drain. That is because it is very easy to *always* keeps the screen on.

Say, for example, you are playing a game. Keeping the screen on while the game is being played is probably a good thing, particularly if the game does not require constant interaction with the screen. However, if you press the in-game pause button, the game might keep the screen on while the game is paused. This might lead you to press pause, put down your tablet (expecting it to fall asleep in a normal period of time), and then have the tablet keep going and going and going... until the battery runs dead.

Whether you use `setKeepScreenOn()` or directly use a `WakeLock`, it is useful to think of three tiers of user interaction.

The first tier is when your app is doing its "one big thing": playing the game, playing the video, displaying the digital book, etc. If you expect that there will be periods of time when the user is actively engaged with your app, but is not interacting with the screen, keep the screen on.

The second tier is when your app is delivering something to the user that *probably* would get used without interaction in the short term, but not indefinitely. For example, a game might reasonably expect that 15 seconds could be too short to have the screen time out, but if the user has not done anything in 5-10 minutes, most likely they are not in front of the game. Similarly, a digital book reader should not try to keep the screen on for an hour without user interaction.

The third tier is when your app is doing anything other than the main content, where normal device behavior should resume. A video player might keep the screen on while the video is playing, but if the video ends, normal behavior should resume. After all, if the person who had been watching the video fell asleep, they will not be in position to press a power button.

The first and third tiers are fairly easy from a programming standpoint. Just `acquire()` and `release()` the `WakeLock`, or toggle `setKeepScreenOn()` between `true` and `false`.

The second tier — where you are willing to have a screen timeout, just not too quickly — requires you to add a bit more smarts to your app. A simple, low-overhead way of addressing this is to have a `postDelayed()` loop, to get a `Runnable` control every 5-10 seconds. Each time the user interacts with your app, update a `lastInteraction` timestamp. The `Runnable` compares `lastInteraction` with the current time, and if it exceeds some threshold, release the `WakeLock` or call `setKeepScreenOn(false)`. When the user interacts again, though, you will need to re-acquire the `WakeLock` or call `setKeepScreenOn(true)`. Basically, you have your *own* inactivity timing mechanism to control when you are inhibiting normal inactivity behavior or not.

To see the second tier in action, take a look at the `MiscUI/DelayedTimeout` sample project.

The UI is a simple button. We want to keep the screen awake while the user is using the button, but let it fall asleep after a period of inactivity that *we* control. To

accomplish this, we will use a `postDelayed()` loop, to get control every 15 seconds to see if there has been user activity:

```
package com.commonsware.android.timeout;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;

public class MainActivity extends Activity implements Runnable {
  private static int TIMEOUT_POLL_PERIOD=15000; // 15 seconds
  private static int TIMEOUT_PERIOD=300000; // 5 minutes
  private View content=null;
  private long lastActivity=SystemClock.uptimeMillis();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    content=findViewById(android.R.id.content);
    content.setKeepScreenOn(true);
    run();
  }

  @Override
  public void onDestroy() {
    content.removeCallbacks(this);

    super.onDestroy();
  }

  @Override
  public void run() {
    if ((SystemClock.uptimeMillis() - lastActivity) > TIMEOUT_PERIOD) {
      content.setKeepScreenOn(false);
    }

    content.postDelayed(this, TIMEOUT_POLL_PERIOD);
  }

  public void onClick(View v) {
    lastActivity=SystemClock.uptimeMillis();
  }
}
```

(from MiscUI/DelayedTimeout/app/src/main/java/com/commonsware/android/timeout/MainActivity.java)

In `onCreate()`, we call `setKeepScreenOn(true)` to keep the screen on, regardless of what the user's default timeout is. Then, we call the `run()` method from our `Runnable` interface (implemented on the activity itself). `run()` sees if 5 minutes has elapsed since the last bit of user activity (initially set to be the time the activity launches). If 5 minutes has elapsed, we revert to normal screen-timeout behavior with `setKeepScreenOn(false)`. We also schedule ourselves, as a `Runnable`, to get control again in 15 seconds, to see if 5 minutes has elapsed since the last-seen activity. Our button's `onClick()` method simply updates the last-seen timestamp, and `onDestroy()` cleans up our `postDelayed()` loop by calling `removeCallbacks()` to stop invoking our `Runnable`.

The net is that the device's screen will remain on for 5 minutes since the last time the user taps the button, even if the user's default screen timeout is set to shorter than 5 minutes. Yet, at the same time, we do not keep the screen on *forever*, causing unnecessary battery drain.

Note that to test this, you will probably need to unplug your USB cable after installing the app on the device (since many developers have it set up to keep the screen on while plugged in). Also, you will need to set your device's screen timeout to be under 5 minutes, if it is not set that way already.

This is a primitive implementation, missing lots of stuff that you would want in production code (e.g., it never calls `setKeepScreenOn(true)` if we flipped it to `false` but *then* tap the button). And the complexity of determining if the user interacted with the screen will be tied to the complexity of your UI.

That being said, by having a more intelligent use of `WakeLock` and `setKeepScreenOn()`, you can deliver value to the user while not accidentally causing excessive battery drain. Users do not always remember to press the power button, so you need to make sure that just because the user made a mistake, that you do not make it worse.

Event Bus Alternatives

Earlier in the book, we covered the concept of an event bus as a way of communicating between portions of our app, focusing on one event bus implementation: greenrobot's EventBus. Later, in the chapter on broadcast `Intent` objects, we briefly covered `LocalBroadcastManager`.

However, those are not the only event buses available for Android, and others may fit your needs better. In this chapter, we will explore these and other event bus implementations, to compare and contrast.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapters on basic event bus usage, broadcast `Intents`, `AlarmManager` and the scheduled service pattern, and `Notifications`.

A Brief Note About the Sample Apps

The sample apps in this chapter are generally designed to run forever.

It is unlikely that you really want them to run forever, though. Hence, please uninstall each sample after experimenting with it, particularly if you are testing on hardware, such as your personal phone. Your battery will appreciate it.

Standard Intents as Event Bus

You can think of the standard `Intent` and `<intent-filter>` system as a three-channel event bus:

- One channel is used for starting activities
- One channel is used for starting or binding to services
- One channel is used for more ad-hoc "broadcast" events

The component starting an activity does not need to communicate directly with code for that activity — in fact, often times this is impossible, as they are separate apps running in separate processes. Instead, the component starting an activity sends an event indicating the particular operation to be performed (e.g., view this URL), and Android and the user determine which of candidate consumers is the one to process that event.

However, broadcast `Intent` objects are a closer analogue to a real "event bus", in that an event produced by somebody can be consumed by zero, one, or several subscribed consumers, based upon the filtering provided by `<intent-filter>` elements in the manifest or `IntentFilter` objects for use with `registerReceiver()`.

In theory, you could use broadcast `Intent` objects as the backbone for a fairly flexible event bus within your app. In practice, this is not usually a good idea:

- Each broadcast involves inter-process communication (IPC), even if the event producer and consumer(s) are in the same process. This adds overhead.
- Because broadcasts are intrinsically IPC, you have to take security into account, to ensure only authorized producers can publish events that the consumers pick up.

However, if you specifically need a cross-process event bus, such as between a suite of related apps, using a broadcast `Intent` is a very likely choice.

LocalBroadcastManager as Event Bus

As was briefly noted earlier in the book, the Android Support package offers a `LocalBroadcastManager`. This is designed to offer an event bus with a feel very similar to classic broadcast `Intent` objects, but local to your process. Not only does this avoid IPC overhead, but it improves security, as other apps have no means of spying on your internal communications.

`LocalBroadcastManager` is supplied by both the `support-v4` and `support-v13` libraries. Generally speaking, if your `minSdkVersion` is less than 13, you probably should choose `support-v4`.

A Simple LocalBroadcastManager Sample

Let's see `LocalBroadcastManager` in action via the `Intents/Local` sample project.

Here, our `LocalActivity` sends a command to a `NoticeService` from `onCreate()`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    notice=(TextView)findViewById(R.id.notice);
    startService(new Intent(this, NoticeService.class));
  }
```

(from Intents/Local/app/src/main/java/com/commonsware/android/localcast/LocalActivity.java)

The `NoticeService` simply delays five seconds, then sends a local broadcast using `LocalBroadcastManager`:

```
package com.commonsware.android.localcast;

import android.app.IntentService;
import android.content.Intent;
import android.os.SystemClock;
import android.support.v4.content.LocalBroadcastManager;

public class NoticeService extends IntentService {
  public static final String BROADCAST=
      "com.commonsware.android.localcast.NoticeService.BROADCAST";
  private static Intent broadcast=new Intent(BROADCAST);

  public NoticeService() {
    super("NoticeService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    SystemClock.sleep(5000);
    LocalBroadcastManager.getInstance(this).sendBroadcast(broadcast);
  }
}
```

(from Intents/Local/app/src/main/java/com/commonsware/android/localcast/NoticeService.java)

Specifically, you get at your process' singleton instance of `LocalBroadcastManager` by calling `getInstance()` on the `LocalBroadcastManager` class.

Our `LocalActivity` registers for this local broadcast in `onStart()`, once again using `getInstance()` on `LocalBroadcastManager`:

```
@Override
public void onStart() {
  super.onStart();

  IntentFilter filter=new IntentFilter(NoticeService.BROADCAST);

  LocalBroadcastManager.getInstance(this).registerReceiver(onNotice,
                                                           filter);
}
```

(from Intents/Local/app/src/main/java/com/commonsware/android/localcast/LocalActivity.java)

`LocalActivity` unregisters for this broadcast in `onStop()`:

```
@Override
public void onStop() {
  super.onStop();

  LocalBroadcastManager.getInstance(this).unregisterReceiver(onNotice);
}
```

(from Intents/Local/app/src/main/java/com/commonsware/android/localcast/LocalActivity.java)

The `BroadcastReceiver` simply updates a `TextView` with the current date and time:

```
private BroadcastReceiver onNotice=new BroadcastReceiver() {
  public void onReceive(Context ctxt, Intent i) {
    notice.setText(new Date().toString());
  }
};
```

(from Intents/Local/app/src/main/java/com/commonsware/android/localcast/LocalActivity.java)

If you start up this activity, you will see a "`(waiting...)`" bit of placeholder text for about five seconds, before having that be replaced by the current date and time.

The `BroadcastReceiver`, the `IntentFilter`, and the `Intent` being broadcast are the same as we would use with full broadcasts. It is merely how we are using them — via `LocalBroadcastManager` – that dictates they are local to our process versus the standard device-wide broadcasts.

A More Elaborate Sample

That sample is not terribly realistic, but it is simple.

A somewhat more realistic sample is the one using `AlarmManager` and `JobIntentService` from earlier in the book. However, that app was also fairly unrealistic, at least in terms of its output, as LogCat is not very useful to users. A more typical approach for a background service like this is to notify a foreground `Activity`, if there is one, about work that was accomplished, and otherwise display a `Notification`. We described that pattern in the chapter on `Notifications`.

In the `EventBus/LocalBroadcastManager` sample project, we blend:

- Having a service wake up every so often to do some work
- Arranging to let the user know of background accomplishments via an `Activity` or a `Notification`
- Using `LocalBroadcastManager` to keep the communications in-process

The Activity

The `EventDemoActivity` that is our app's entry point is a bit similar to the one used in the `AlarmManager` demo, in that it calls `scheduleAlarms()` on `PollReceiver` to set up the `AlarmManager` schedule:

```
package com.commonsware.android.eventbus.lbm;

import android.support.v4.app.FragmentActivity;
import android.os.Bundle;

public class EventDemoActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
      getSupportFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new EventLogFragment()).commit();

      PollReceiver.scheduleAlarms(this);
    }
  }
}
```

(from EventBus/LocalBroadcastManager/app/src/main/java/com/commonsware/android/eventbus/lbm/EventDemoActivity.java)

However, we also put an `EventLogFragment` on the screen, if it is not already there, via a `FragmentTransaction`. This is where we will display events coming from the service, while our activity is in the foreground. We will examine `EventLogFragment` and how it participates in the event bus shortly.

The PollReceiver

`PollReceiver` is largely unchanged from its `AlarmManager` demo original edition. This `BroadcastReceiver` will be used both for getting control at boot time (to reschedule the alarms, wiped on the reboot) and for sending the work to the `ScheduledService` for processing:

```
package com.commonsware.android.eventbus.lbm;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class PollReceiver extends BroadcastReceiver {
  private static final int PERIOD=60000; // 1 minute
  private static final int INITIAL_DELAY=5000; // 5 seconds

  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() == null) {
      ScheduledService.enqueueWork(ctxt);
    }
    else {
      scheduleAlarms(ctxt);
    }
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime() + INITIAL_DELAY,
                     PERIOD, pi);
```

```
  }
}
```

(from EventBus/LocalBroadcastManager/app/src/main/java/com/commonsware/android/eventbus/lbm/PollReceiver.java)

ScheduledService and Sending Events

Before, our `ScheduledService` just dumped a message to LogCat. This was crude but effective for what that demo required. Now, we want our service to let the UI layer know about some work that was accomplished, or to raise a `Notification`.

In this case, the "work" is generating a random number.

```
package com.commonsware.android.eventbus.lbm;

import android.app.Notification;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.support.v4.app.JobIntentService;
import android.support.v4.app.NotificationCompat;
import android.support.v4.content.LocalBroadcastManager;
import java.util.Calendar;
import java.util.Random;

public class ScheduledService extends JobIntentService {
  private static int NOTIFY_ID=1337;
  private static final int UNIQUE_JOB_ID=1337;
  private static final String CHANNEL_WHATEVER="channel_whatever";
  private Random rng=new Random();

  static void enqueueWork(Context ctxt) {
    enqueueWork(ctxt, ScheduledService.class, UNIQUE_JOB_ID,
      new Intent(ctxt, ScheduledService.class));
  }

  @Override
  public void onHandleWork(Intent i) {
    Intent event=new Intent(EventLogFragment.ACTION_EVENT);
    long now=Calendar.getInstance().getTimeInMillis();
    int random=rng.nextInt();

    event.putExtra(EventLogFragment.EXTRA_RANDOM, random);
    event.putExtra(EventLogFragment.EXTRA_TIME, now);

    if (!LocalBroadcastManager.getInstance(this).sendBroadcast(event)) {
      NotificationManager mgr=
        (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

      if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
        mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
        mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
```

```
          "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
      }

      NotificationCompat.Builder b=new NotificationCompat.Builder(this, CHANNEL_WHATEVER);
      Intent ui=new Intent(this, EventDemoActivity.class);

      b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
       .setContentTitle(getString(R.string.notif_title))
       .setContentText(Integer.toHexString(random))
       .setSmallIcon(android.R.drawable.stat_notify_more)
       .setTicker(getString(R.string.notif_title))
       .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

      mgr.notify(NOTIFY_ID, b.build());
    }
  }
}
```

(from EventBus/LocalBroadcastManager/app/src/main/java/com/commonsware/android/eventbus/lbm/ScheduledService.java)

`LocalBroadcastManager`, as we have seen, uses the same `Intent` and `IntentFilter` and `BroadcastReceiver` structures as are used with regular broadcasts, just via a singleton message bus (`LocalBroadcastManager.getInstance()`) instead of the framework's IPC engine. Hence, we need an `Intent` that represents the message, so we create one, using an action string published by the `EventLogFragment`. We also attach two extras to this `Intent`, using keys published by `EventLogFragment`: the random number, plus the time of this event.

We then call `sendBroadcast()` on the singleton `LocalBroadcastManager`. This returns a `boolean` value, `true` indicating that one or more locally-registered receivers were delivered the `Intent`, `false` otherwise. Hence, if `sendBroadcast()` returns `true`, we can assume that somebody in the UI layer picked up our message and is now responsible for displaying these results to the user.

Conversely, if `sendBroadcast()` returns `false`, we must assume that the UI layer did not receive the message, and so the service should inform the user directly, in this case via a `Notification`, showing the random number as the text in the notification drawer.

EventLogFragment and Receiving Events

`EventLogFragment`, therefore, is responsible for:

- Registering (and unregistering) to receive the broadcasts to be sent locally by the service
- Doing something with those events to inform the user about the all-important random numbers

In this case, we use a retained `ListFragment` with a `ListView` set into transcript mode, meaning that entries are added at the bottom, and older entries scroll off the top, like a chat transcript:

```
package com.commonsware.android.eventbus.lbm;

import android.annotation.SuppressLint;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.support.v4.content.LocalBroadcastManager;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Locale;

public class EventLogFragment extends ListFragment {
  static final String EXTRA_RANDOM="r";
  static final String EXTRA_TIME="t";
  static final String ACTION_EVENT="e";
  private EventLogAdapter adapter=null;

  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    setRetainInstance(true);
    getListView().setTranscriptMode(ListView.TRANSCRIPT_MODE_NORMAL);

    if (adapter == null) {
      adapter=new EventLogAdapter();
    }

    setListAdapter(adapter);
  }

  @Override
  public void onStart() {
```

```
    super.onStart();

    IntentFilter filter=new IntentFilter(ACTION_EVENT);

    LocalBroadcastManager.getInstance(getActivity())
                         .registerReceiver(onEvent, filter);
  }

  @Override
  public void onStop() {
    LocalBroadcastManager.getInstance(getActivity())
                         .unregisterReceiver(onEvent);

    super.onStop();
  }

  class EventLogAdapter extends ArrayAdapter<Intent> {
    DateFormat fmt=new SimpleDateFormat("HH:mm:ss", Locale.US);

    public EventLogAdapter() {
      super(getActivity(), android.R.layout.simple_list_item_1,
            new ArrayList<Intent>());
    }

    @SuppressLint("DefaultLocale")
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      TextView row=
          (TextView)super.getView(position, convertView, parent);
      Intent event=getItem(position);
      Date date=new Date(event.getLongExtra(EXTRA_TIME, 0));

      row.setText(String.format("%s = %x", fmt.format(date),
                                event.getIntExtra(EXTRA_RANDOM, -1)));

      return(row);
    }
  }

  private BroadcastReceiver onEvent=new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
      adapter.add(intent);
    }
  };
}
```

(from EventBus/LocalBroadcastManager/app/src/main/java/com/commonsware/android/eventbus/lbm/EventLogFragment.java)

The `ListAdapter` for the `ListView` is an `EventLogAdapter`, an `ArrayAdapter` for `Intent` objects, where in `getView()` we populate the list rows with the time and random value.

In `onStart()` and `onStop()`, we register for (and unregister from) the desired broadcast, pointing to an `onEvent` `BroadcastReceiver` that adds the incoming `Intent` to the `EventLogAdapter`. That, in turn, updates the `ListView`.

The result is that while the activity is in the foreground, the events will be displayed to the user directly:

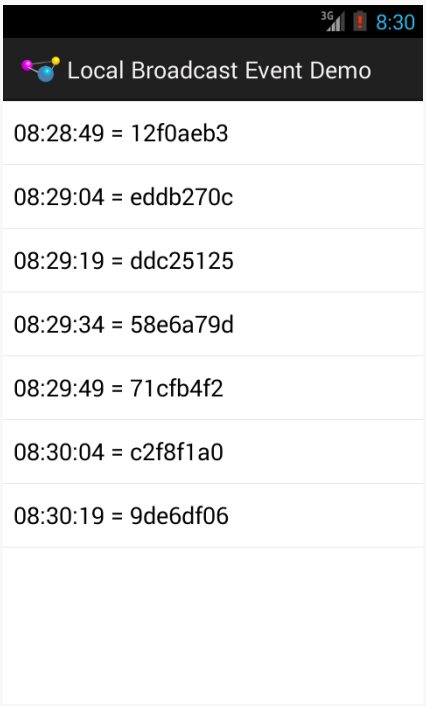


Figure 68o: LocalBroadcastManager as Event Bus, Demo Activity

Whereas if events are processed while the activity is not in the foreground, a `Notification` will be shown with the last results:

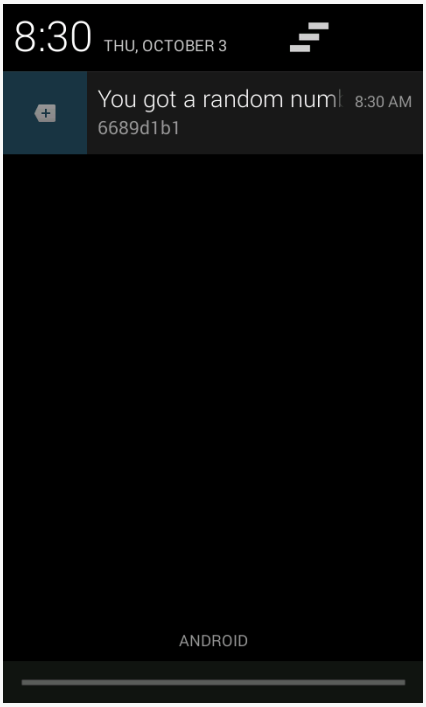


Figure 681: LocalBroadcastManager as Event Bus, Demo Notification

Reference, Not Value

When you send a "real" broadcast `Intent`, your `Intent` is converted into a byte array (courtesy of the `Parcelable` interface) and transmitted to other processes. This occurs even if the recipient of the `Intent` is within your own process — that is what makes `LocalBroadcastManager` faster, as it avoids the inter-process communication.

However, since `LocalBroadcastManager` does not need to send your `Intent` between processes, that means it does not turn your `Intent` into a byte array. Instead, it just passes the `Intent` along to any registered `BroadcastReceiver` with a matching `IntentFilter`. In effect, while "real" broadcasts are pass-by-value, local broadcasts are pass-by-reference.

This can have subtle side effects.

For example, there are a few ways that you can put a collection into an `Intent` extra, such as `putStringArrayListExtra()`. This takes an `ArrayList` as a parameter. With a real broadcast, once you send the broadcast, it does not matter what happens to the original `ArrayList` — the rest of the system is working off of a copy. With a local broadcast, though, the `Intent` holds onto the `ArrayList` you supplied via the setter.

If you change that `ArrayList` elsewhere (e.g., clear it for reuse), the recipient of the `Intent` will see those changes.

Similarly, if you put a `Parcelable` object in an extra, the `Intent` holds onto the actual object while it is being broadcast locally, whereas a real broadcast would have resulted in a copy. If you change the object while the broadcast is in progress, the recipient of the broadcast will see those changes.

This can be a feature, not a bug, when used properly. But, regardless, it is a non-trivial difference, one that you will need to keep in mind.

Limitations of Local

While `LocalBroadcastManager` is certainly useful, it has some serious limitations.

The biggest is that it is purely local. While traditional broadcasts can either be internal (via `setPackage()`) or device-wide, `LocalBroadcastManager` only handles the local case. Hence, anything that might involve other processes, such as a `PendingIntent`, will not use `LocalBroadcastManager`. For example, you cannot register a receiver through `LocalBroadcastManager`, then use a `getBroadcast()` `PendingIntent` to try to reach that `BroadcastReceiver`. The `PendingIntent` will use the regular broadcast `Intent` mechanism, which the local-only receiver will not respond to.

Similarly, since a manifest-registered `BroadcastReceiver` is spawned via the operating system upon receipt of a matching true broadcast, you cannot use such receivers with `LocalBroadcastManager`. Only a `BroadcastReceiver` registered via `registerReceiver()` on the `LocalBroadcastManager` will use the `LocalBroadcastManager`. For example, you cannot implement the `Activity`-or-`Notification` pattern that we will see later in this book via `LocalBroadcastManager`.

Also, `LocalBroadcastManager` does not offer ordered or sticky broadcasts.

greenrobot's EventBus 3.x

`LocalBroadcastManager` has two major advantages:

1. It is part of the Android Support package, and therefore it is part of the officially-supported corner of the Android ecosystem

2. It works like traditional broadcasts, which will make it easier for some developers to "wrap their heads around" it

However, that same dependency on the `Intent` and `IntentFilter` structure adds bulk and limits flexibility. Hence, it is not surprising that there are alternative event buses to `LocalBroadcastManager`.

Java, outside of Android, has had a few event bus implementations. One of the more popular ones in recent years has been the event bus that is part of Google's Guava family of libraries. However, while a Java event bus perhaps can be used on Android, it may not be *optimal* for Android. Hence, a few projects have started with Guava's event bus implementation and have extended it to be a bit more Android-aware, or perhaps even Android-centric.

greenrobot's EventBus is one such event bus.

NOTE: For the purposes of this chapter, "greenrobot's EventBus" refers to the library, and "`EventBus`"" refers to the `EventBus` Java class in that library.

Basic Usage and Sample App

With `LocalBroadcastManager`, you work with a singleton instance, calling methods like `registerReceiver()` and `sendBroadcast()` upon it to subscribe to and raise events, respectively.

With greenrobot's EventBus, you work with an `EventBus` instance, calling methods like `register()` and `post()` upon it to subscribe to and raise events, respectively. Usually, we use the singleton instance of `EventBus` that we get by calling `getDefault()` on the `EventBus` class, but you are welcome to have different `EventBus` objects, representing distinct communications channels, if you wish.

Hence, at the core, greenrobot's EventBus behaves much like `LocalBroadcastManager`. What differs is in the nature of the events and the subscribers.

With `LocalBroadcastManager`, events are `Intent` objects. With greenrobot's EventBus, an event can be whatever data type you like. Hence, you can create your own `...Event` classes, holding whatever bits of data, in whatever data types suit you — you are not restricted to things that can go in an `Intent` extra. However, as has been noted on occasion, "with great power comes great responsibility", and so you will need to ensure that you use this carefully and do not wind up creating some sort

of memory leak as a result. For example, do not pass something from an `Activity` to a `Service` via a custom event, where the `Service` will hold onto that information for a long time, if that "something" holds a reference back to the `Activity`.

With `LocalBroadcastManager`, subscribers are `BroadcastReceivers`, who use an `IntentFilter` to identify which events they are interested in. With greenrobot's EventBus, subscribers are any class you want. A special `@Subscribe` annotation is used to both indicate what sorts of events the subscriber is interested in (based on the parameter to the annotated method) and what method should be invoked when a matching event is raised (the annotated method itself). Hence, not only do you use custom event classes to allow you to carry along custom data, but you use them as a filtering mechanism, much like you would use custom action strings with `LocalBroadcastManager`.

To see how this works, take a look at the `EventBus/GreenRobot3` sample project, which is a clone of the `EventBus/LocalBroadcastManager` demo, but one where we substitute in greenrobot's EventBus as a replacement for `LocalBroadcastManager`. Our activity and `PollReceiver` are unchanged: they did not directly interact with `LocalBroadcastManager` and do not need to interact with greenrobot's EventBus. The changes are isolated in our `ScheduledService` and `EventLogFragment`.

ScheduledService and Sending Events

We will need an `EventBus` instance, one that serves the same basic role as does the singleton `LocalBroadcastManager` retrieved by `getInstance()`. As noted above, you can call `getDefault()` on `EventBus` to get a singleton `EventBus` instance, and this suffices in most cases.

When it comes time for us to send a message, we can call `post()` on the `EventBus`, supplying whatever sort of event object that we want:

```
      EventBus.getDefault().post(randomEvent);
```

(from EventBus/GreenRobot3/app/src/main/java/com/commonsware/android/eventbus/greenrobot/ScheduledService.java)

Here, we are posting an instance of a `RandomEvent`:

```
package com.commonsware.android.eventbus.greenrobot;

import java.util.Calendar;
import java.util.Date;
```

```
public class RandomEvent {
  Date when=Calendar.getInstance().getTime();
  int value;

  RandomEvent(int value) {
    this.value=value;
  }
}
```

(from EventBus/GreenRobot3/app/src/main/java/com/commonsware/android/eventbus/greenrobot/RandomEvent.java)

EventLogFragment and Receiving Events

Over in our `EventLogFragment`, rather than register and unregister a `BroadcastReceiver` in `onStart()` and `onStop()`, we register and unregister the fragment itself with the `EventBus`:

```
  @Override
  public void onStart() {
    super.onStart();

    EventBus.getDefault().register(this);
  }

  @Override
  public void onStop() {
    EventBus.getDefault().unregister(this);

    super.onStop();
  }
```

(from EventBus/GreenRobot3/app/src/main/java/com/commonsware/android/eventbus/greenrobot/EventLogFragment.java)

Now, we can use the `@Subscribe` annotation to arrange to receive any event we want that is delivered via this `EventBus`, based on event class. Since we want to receive `RandomEvent` messages, we merely need to have a `public void` method, taking a `RandomEvent` parameter, marked with the `@Subscribe` annotation, such as `onRandomEvent()`:

```
  @Subscribe(threadMode = ThreadMode.MAIN)
  public void onRandomEvent(final RandomEvent event) {
    adapter.add(event);
  }
```

(from EventBus/GreenRobot3/app/src/main/java/com/commonsware/android/eventbus/greenrobot/EventLogFragment.java)

Note that the method name can be anything we want, as it is the annotation, not the method name, that identifies this as being an event handling method.

Since Java annotations can take key-value pairs for configuration, EventBus 3.x uses that to configure the behavior of `@Subscribe`. Here, we use `@Subscribe(threadMode = ThreadMode.MAIN)`, to indicate that we want this event to be delivered to this method on the main application thread.

In this method, we can do what we need to with our `RandomEvent`. In our case, `EventLogAdapter` has been modified to be an `ArrayAdapter` of `RandomEvent`, as opposed to being an `ArrayAdapter` of `Intent` as in the earlier sample. What we want to do is append the new `RandomEvent` to the end of the adapter.

Handling the "Nobody's Home" Scenario

What is missing, though, is the logic we used in `LocalBroadcastManager` to determine if somebody received our message, where we raised a `Notification` if that is not the case.

The solution for this with greenrobot's EventBus is to call `hasSubscriberForEvent()`, with the Java `Class` object of the event that we would like to `post()`. If this returns `true`, we have a current subscriber; otherwise, we do not.

So, the full `onHandleWork()` implementation uses `hasSubscriberForEvent()` and either uses `post()` to raise the event or displays a `Notification` itself:

```
@Override
public void onHandleWork(Intent i) {
  RandomEvent randomEvent=new RandomEvent(rng.nextInt());

  if (EventBus.getDefault().hasSubscriberForEvent(randomEvent.getClass())) {
    EventBus.getDefault().post(randomEvent);
  }
  else {
    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    NotificationCompat.Builder b=new NotificationCompat.Builder(this, CHANNEL_WHATEVER);
    Intent ui=new Intent(this, EventDemoActivity.class);
```

```
    b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
      .setContentTitle(getString(R.string.notif_title))
      .setContentText(Integer.toHexString(randomEvent.value))
      .setSmallIcon(android.R.drawable.stat_notify_more)
      .setTicker(getString(R.string.notif_title))
      .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

    mgr.notify(NOTIFY_ID, b.build());
  }
}
```

(from EventBus/GreenRobot3/app/src/main/java/com/commonsware/android/eventbus/greenrobot/ScheduledService.java)

There is a race condition, though. Since our `hasSubscriberForEvent()` call is happening on a background thread, it is possible that between the `hasSubscriberForEvent()` call and the `post()` call that the subscriber unsubscribes. This is an unlikely occurrence here, but it is worth keeping in mind.

Other Notable Capabilities

In addition to the threading features, greenrobot's EventBus has a few other noteworthy bells and whistles:

- Other thread modes are available, including `ThreadMode.POSTING` (events are delivered on the same thread they are posted from) and `ThreadMode.BACKGROUND` (events are delivered on a background thread, with `EventBus` using its own thread if the event was posted from the main application thread).
- `postSticky()` and `registerSticky()` allow you to have sticky events, much like sticky broadcasts with the classic broadcast `Intent` system.
- Ordered event processing as an option, akin to ordered broadcasts. You accomplish this by assigning a priority to the event handling method (e.g., `@Subscribe(priority = 1)`). If a higher-priority handler wants to consume the event, it can call `cancelEventDelivery()` on the `EventBus`, passing in the event object.

Hey, What About Otto?

For a few years, a third major event bus implementation was popular: Square's Otto. Like greenrobot's EventBus, Otto was based off of Guava's `EventBus` class and was tuned towards Android app development. It shared some characteristics with greenrobot's EventBus, owing to the shared heritage. On the whole, greenrobot's EventBus was a bit more complex to use but offered greater flexibility.

Square has since discontinued work on Otto, so unless you have existing legacy code that uses Otto, you should use some other event bus implementation.

Tasks

One of the most confusing aspects of Android to deal with is the concept of tasks. Fortunately, the automatic management of tasks is *almost* enough to get by, without you having to do much customization. However, many apps will have needs to tailor how their app interacts with the task system, and understanding what is possible and how to do it is not easy. It is made even more complicated by changes to Android, from both engineering and design perspectives, over the years.

This chapter will attempt to untie the knot of knowledge surrounding Android's task system, explaining why things are the way they are. However, there will be a few places where the knot turns a bit Gordian, and we will have to settle for more about "how" and less about "why" the task system works as it does.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

One sample app makes heavy use of the `PackageManager` system service and refers in a few places to the `Launchalot` sample app profiled in that chapter.

First, Some Terminology

It will be useful to establish some common definitions of terms that you will encounter, both in this chapter and in other materials that describe the task system.

Task

So, what exactly *is* a "task"?

The Android developer documentation describes it as:

> A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the back stack), in the order in which each activity is opened.

In that sense, a task is reminiscent of a tab in a tabbed browser. As the user navigates, clicking links and submitting forms, the user advances into other Web pages. Those pages could be on the same site as they started or could be on different sites. The browser BACK button is supposed to reverse the navigation, allowing the user to return from whence they came.

Back Stack

The user perceives tasks mostly in the form of pressing the BACK button, using this to return to previous "screens" that they had been on previously.

Sometimes, BACK button processing is handled within a single activity, such as when you put a dynamic fragment onto the "back stack" via `addToBackStack()` on a `FragmentTransaction`. Or, the activity could override `onBackPressed()` and do special stuff in certain scenarios. Those are part of the user experience of pressing BACK. From the standpoint of the task system, though, internal consumption of the BACK button presses do not affect the task.

At the task level, the "back stack" refers to a chain of activities. This matches the behavior of Web sites, where while pressing the browser BACK button *might* trigger in-page behavior, usually it returns you to the previous page. Similarly, while pressing BACK on an Android device *might* trigger in-activity behavior, usually it triggers a call to `finish()` on the foreground activity and returns control to whatever had preceded it on the back stack.

Recent Tasks

In a tabbed Web browser, if we have several tabs open, we think of all of them as being "running". Frequently, we do not really even think about the concept, any more than we might think about the state of tabs in an IDE other than the one that

we are working in right now. However, if you have ever had some browser tab all of a sudden start playing audio, such as from a reloaded page pulling in an audio-enabled ad banner, you are well aware that tabs are "running", while you are also "running" to try to figure out what tab is playing the audio so you can get rid of it.

However, that is the behavior on a desktop Web browser. A desktop Web browser is not subject to heap size limitations the way Android apps are. And, historically, mobile devices had less system RAM than did their desktop and notebook counterparts, though that is rapidly changing.

In Android, therefore, developers are used to the notion that their processes may be terminated, while in the background, to free up memory for other processes. This is being done to allow for more apps to deliver more value in less system RAM.

However, from a multitasking standpoint, having apps just up and vanish is awkward. Hence, Android has the notion of "recent tasks". These are tasks, with their corresponding back stacks, that the user has been in "recently". How far back "recently" goes depends a bit on the version of Android – there could be as few as eight items. These "recent tasks" may or may not have a currently-running *process* associated with them. However, if the user chooses to return to one of those recent tasks, and there is no process for it, Android will seamlessly fork a fresh process, to be able to not only start up those apps, but return the user to where they were, in terms of UI contents (e.g., saved instance state `Bundle`) and in terms of back stack contents (e.g., where the user goes if the user now presses BACK).

Overview Screen

In a tabbed Web browser, you can navigate between different tabs in some browser-specific way. Some tabs may have the actual "tab" visible around the address bar. Some tabs might only be reachable via some sort of scrolling operation, or via a drop-down list, for people who have lots and lots of tabs open. Regardless, there is some UI means to pick the tab that you want to be viewing in the main browser area.

In Android, the "overview screen" is where the user can view the recent tasks and choose to return to one of them. Many people, including this author, refer to this as the "recent tasks list", but apparently the official term is "overview screen".

The way the overview screen has looked and worked has changed over the years.

Android 1.x/2.x

In the early days of Android, long-pressing the HOME button would bring up the overview screen, with up to eight recent tasks:

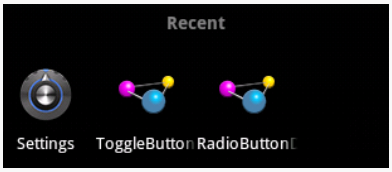


Figure 682: Overview Screen, from Android 2.3.3

And... that was pretty much it.

Android 3.x/4.x

The overall move to the holographic theme for Android brought with us a new icon, for a dedicated way to get to the overview screen:

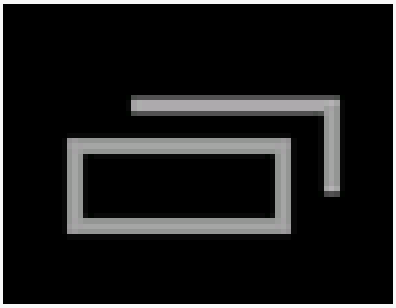


Figure 683: Overview Screen/Recent Tasks Navigation Bar Icon, from Android 4.3

Devices that offered a navigation bar at the bottom would have this button. Devices that chose to have off-screen affordances for BACK and HOME might have a similar button for the overview screen. For those that neither had a navigation bar nor a dedicated off-screen button for the overview screen, long-pressing HOME would bring up the overview screen.

The overview screen could have more apps (15 or so) before old tasks would be dropped:

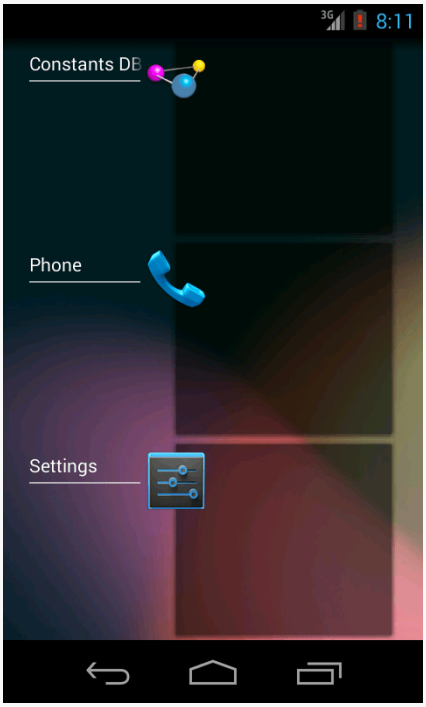


Figure 684: Overview Screen, from Android 4.3

The overview screen also added more improvements:

- Thumbnails of the top activity in each task's back stack, except for those activities that used `FLAG_SECURE` to block this, and except on some emulator images
- Swiping an entry off the list would remove that recent task

Android 5.x

Functionally, the Android 5.x overview screen functions much like its 4.x counterpart, with the ability to see previews of tasks and remove tasks from the screen.

However, there are some differences, starting with the navigation bar icon used to bring up the overview screen:

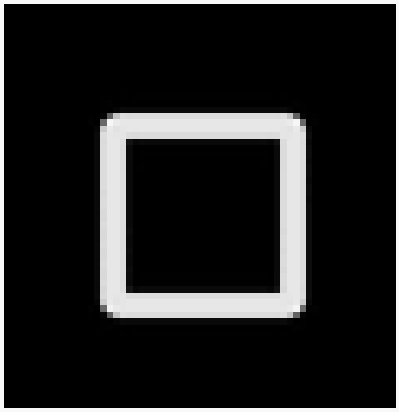


Figure 685: Overview Screen/Recent Tasks Navigation Bar Icon, from Android 5.0

Also, the previews are larger and stacked like cards, more so than being a classically vertically-scrolling list:

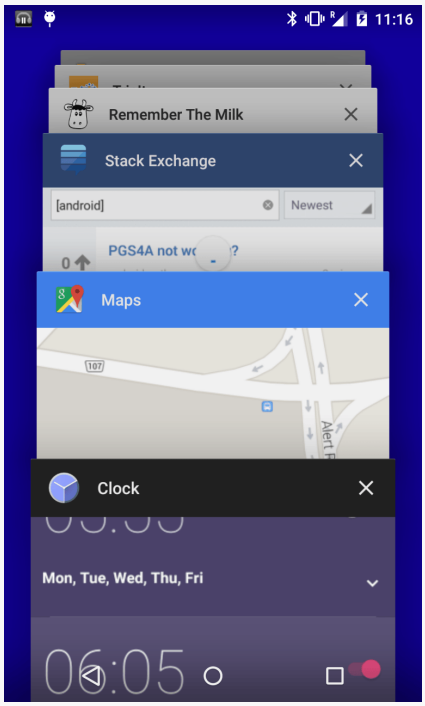


Figure 686: Overview Screen, from Android 5.0

More importantly:

- The roster of recent tasks will be restored after a reboot, and

- If there is a limit on how many entries can appear in the list, the author has not run into it yet

Running Tasks

A running task is a task that has running process(es) associated with it. Recent tasks may or may not be running.

And Now, a Bit About Task Killers

In October 2008, the first Android device was publicly released (the T-Mobile G1, a.k.a., HTC Dream).

Around December of 2008, the first task killers appeared on the Android Market (now the Play Store).

While the techniques used in 2008 to kill tasks were removed in later releases, some amount of task management behavior still exists in Android. Having a task killer is useful for understanding how tasks (and their killers) behave on Android. In particular, it is useful to have a way to emulate an app's process being terminated due to low memory conditions... which is exactly what modern task killers do.

So, in this section, we will explore the concept of task killers, including how to implement one, before using this tool to help us explore the overall Android task system.

What Do Task Killers Do?

Despite the name, task killers do not kill tasks.

Rather, task killers terminate background processes. This does not impact the task, insofar as it will still be in the recent tasks roster and will still show up on the overview screen. However, the process for the app associated with the task will shut down.

Task killers can only request to terminate background processes. If your app is in the foreground (i.e., has the foreground activity), you cannot be terminated by a task killer.

To terminate background processes, task killers need to hold the `KILL_BACKGROUND_PROCESSES` permission, via a `<uses-permission>` element in their manifests. That enables them to be able to call the `killBackgroundProcesses()` method on `ActivityManager`. Supplied an application ID, `killBackgroundProcesses()` will terminate any background process(es) associated with that application. Normally, there will only be one such process, but if the app in question is using the `android:process` attribute in the manifest to have multiple processes, then all the app's processes will be terminated.

This termination is done using the same internal mechanism that is used by the "out-of-memory killer", which is responsible for freeing up system RAM due to low memory conditions.

Killing vs. Force-Stopping

For ordinary users, there are a few options for terminating background processes. Using a task killer, or swiping the task off the overview screen on Android 4.0+, will terminate background processes. Both use `killBackgroundProcesses()` (or internal equivalents).

However, users can also go into the Settings app, find the app in the list of installed apps, and click a "Force Stop" button associated with that app. On the surface, this has a similar effect to the above techniques, as the background process is terminated. However, force-stopping the app also unschedules any `AlarmManager` or `JobScheduler` events for that app, plus moves the app back into the "stopped state", blocking manifest-registered broadcast receivers. Hence, force-stopping an app has a much larger impact than does merely using a task killer.

A few devices have manufacturer-supplied task managers (a.k.a., task killers), where stopping an app from those apps actually does a force stop behind the scenes, rather than `killBackgroundProcesses()`. This is not a good idea, as force-stopping an app has the aforementioned side effects. Fortunately, third-party task killers cannot force-stop apps, barring any security flaws in Android that might make this possible.

Why Use One?

Nowadays, normally, users do not need task killers. Occasionally one can be useful, to stop a background process for a poorly-written app (e.g., one that powers on GPS but fails to let go of GPS when the app moves to the background). On most modern Android devices, swiping the app off the overview screen usually suffices, and so

task killers are not nearly as crucial as they were in Android 1.x/2.x, where there was no such built-in background process management solution.

For developers, the problem with swiping an app off the overview screen is that it not only terminates background processes, but it also removes the task entirely. This makes it difficult to see what the behavior is when apps' processes terminate for more conventional reasons (e.g., out-of-memory killer) and how tasks tie into that. While developers have the ability to stop processes through development tools (e.g., the process list in DDMS), that just terminates the process, and it may do so slightly differently than does the out-of-memory killer. Hence, having a task killer around can be useful for experimentation purposes.

And, since getting a task killer on an emulator can be challenging (since emulators do not have access to the Play Store), having the source code for a simple task killer is useful for developers. So, let's look at how to implement a task killer.

A Killer Sample

The `Tasks/Nukesalot` sample application implements the Nukesalot app. This is a reworked version of the Launchalot sample from elsewhere in the book. Launchalot lists the launchable activities and lets the user launch those activities by clicking on them in a `ListView`. Nukesalot lists the running applications and allows the user to kill those applications' background processes by clicking on them in a `ListView`.

Finding Killable Apps

First, we need to find apps that are eligible to be killed.

In `onCreate()` of the `MainActivity`, we get our hands on an `ActivityManager` system service. `ActivityManager` is not strictly tied to the UI construct known as activities, but rather to general "activity" of the user with respect to the device.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  am=(ActivityManager)getSystemService(ACTIVITY_SERVICE);
}
```

(from Tasks/Nukesalot/app/src/main/java/com/commonsware/android/nukesalot/Nukesalot.java)

In `onResume()`, we then call a private `buildAdapter()` method to create an instance of an `AppAdapter` for us:

```
@Override
public void onResume() {
  super.onResume();

  adapter=buildAdapter();
  setListAdapter(adapter);
}
```

(from Tasks/Nukesalot/app/src/main/java/com/commonsware/android/nukesalot/Nukesalot.java)

We do this in `onResume()` so that when our activity returns to the foreground, it shows a fresh list of running apps. This activity lacks the ability to detect new running processes on the fly, something that could be addressed by a manual refresh option (e.g., action bar item). The implementation of this is left as an exercise for the reader.

`buildAdapter()` needs to find out the application ID (a.k.a., package name) of the running applications. It would be easier to list all applications, but there is little point in listing apps that cannot be killed simply because they are not running. The roster of application IDs of the running apps is a `HashSet` named `runningPackages`, initially empty:

```
private AppAdapter buildAdapter() {
  HashSet<String> runningPackages=new HashSet<String>();

  for (ActivityManager.RunningAppProcessInfo proc :
      am.getRunningAppProcesses()) {
    for (String pkg : proc.pkgList) {
      runningPackages.add(pkg);
    }
  }

  PackageManager pm=getPackageManager();
  List<ApplicationInfo> apps=new ArrayList<ApplicationInfo>();

  for (ApplicationInfo app : pm.getInstalledApplications(0)) {
    if (runningPackages.contains(app.packageName)) {
      apps.add(app);
    }
  }

  Collections.sort(apps,
      new ApplicationInfo.DisplayNameComparator(pm));
```

```
    return(new AppAdapter(pm, apps));
  }
```

(from Tasks/Nukesalot/app/src/main/java/com/commonsware/android/nukesalot/Nukesalot.java)

We then iterate over the running application processes, obtained via a call to `getRunningAppProcesses()` on the `ActivityManager`. In theory (though it is unclear how this works in practice), a running app process could hold code from multiple packages. We find out those packages via the `pkgList` in each `RunningAppProcessInfo` object that we get back from `getRunningAppProcesses()`. We then iterate over the strings in `pkgList` and add each to `runningPackages`.

Next, in order to display icons and names for these apps, we really need `ApplicationInfo` objects for each app. Plus, it would be nice if these were in some logical order. So, we get a `PackageManager`, create an `ArrayList` of `ApplicationInfo` objects named `apps`, and iterate over all installed applications (via `getInstalledApplications()` on `PackageManager`). For each app, if its package name (via the `packageName` attribute on the `ApplicationInfo`) is in our `runningPackages`, we add the `ApplicationInfo` to the `ArrayList`. We then sort the `ArrayList` using an `ApplicationInfo.DisplayNameComparator` convenience class provided by the Android SDK, which will sort `ApplicationInfo` arrays based on the display name. We then wrap the `ApplicationInfo` list in an `AppAdapter` and return it.

Displaying Killable Apps

`AppAdapter` itself is an `ArrayAdapter` for `ApplicationInfo` objects, designed to render them in rows containing the app's icon and display name:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  >
  <ImageView android:id="@+id/icon"
    android:layout_width="48dp"
    android:layout_height="48dp"
    android:layout_alignParentLeft="true"
    android:layout_margin="2dp"
    android:scaleType="fitCenter"
    android:layout_gravity="center_vertical"
  />
```

```
  <TextView
    android:id="@+id/label"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="24sp"
    android:layout_margin="2dp"
    android:layout_gravity="center_vertical"
  />
</LinearLayout>
```

(from Tasks/Nukesalot/app/src/main/res/layout/row.xml)

`AppAdapter` steals a page from `CursorAdapter` and has `getView()` delegate to `newView()` and `bindView()` methods. `newView()` is called when there is no row to recycle, and it just inflates the `row` layout. `bindView()` uses `PackageManager` to populate the icon and display name widgets using `loadIcon()` and `loadLabel()` calls:

```
  class AppAdapter extends ArrayAdapter<ApplicationInfo> {
    private PackageManager pm=null;

    AppAdapter(PackageManager pm, List<ApplicationInfo> apps) {
      super(Nukesalot.this, R.layout.row, apps);
      this.pm=pm;
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      if (convertView==null) {
        convertView=newView(parent);
      }

      bindView(position, convertView);

      return(convertView);
    }

    private View newView(ViewGroup parent) {
      return(getLayoutInflater().inflate(R.layout.row, parent, false));
    }

    private void bindView(int position, View row) {
      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(getItem(position).loadLabel(pm));
```

```
      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      icon.setImageDrawable(getItem(position).loadIcon(pm));
    }
  }
```

(from Tasks/Nukesalot/app/src/main/java/com/commonsware/android/nukesalot/Nukesalot.java)

`loadIcon()` and `loadLabel()` are methods on `ApplicationInfo` that, given a `PackageManager`, can find the proper resources for those items and retrieve them from the foreign app's package.

The result is a `ListView` filled with running apps... including Nukesalot itself:

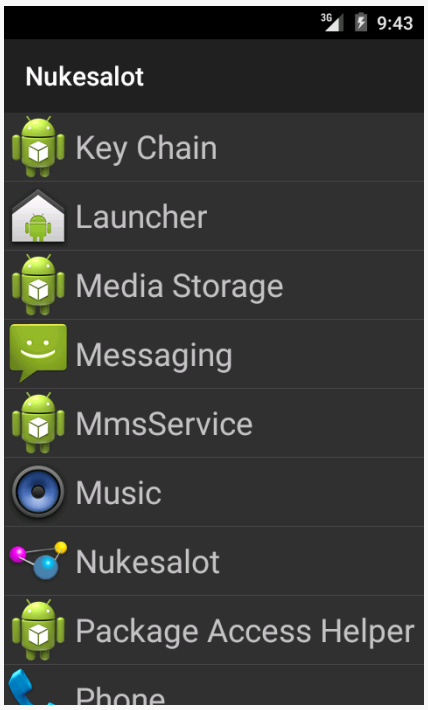


Figure 687: Nukesalot, on Android 5.0

That App Needed Killin'

The idea is that when the user taps on a row in the list, Nukesalot will go and terminate that app's process. So, in `onListItemClick()`, we determine the `ApplicationInfo` of the clicked-upon app, and call `killBackgroundProcesses()` on that app's package name. Then, we refresh the adapter, to show the updated roster of running apps:

```
@Override
protected void onListItemClick(ListView l, View v,
                               int position, long id) {
  ApplicationInfo app=adapter.getItem(position);

  am.killBackgroundProcesses(app.packageName);
  adapter=buildAdapter();
  setListAdapter(adapter);
}
```

(from Tasks/Nukesalot/app/src/main/java/com/commonsware/android/nukesalot/Nukesalot.java)

`killBackgroundProcesses()` requires the `KILL_BACKGROUND_PROCESSES` permission, which we have in the manifest:

```
<uses-permission android:name="android.permission.KILL_BACKGROUND_PROCESSES"/>
```

If you tap on a row for some ordinary Android app, other than Nukesalot itself, that process will be terminated. However:

- Nukesalot cannot kill itself, as it is not a background process, but rather is the foreground process. In principle, we could detect this case, `finish()` the activity, and fork a background thread to call `killBackgroundProcesses()` after a short delay to ensure that our process is categorized as a background process. In practice, that seems like an awful lot of work for a book example.
- Some system processes (e.g., "Android System") simply cannot be killed using `killBackgroundProcesses()`.

More importantly, from the standpoint of this chapter, is that killing a background process using Nukesalot does not disturb the recent-tasks list. Our task still shows up there, even though we terminated the process for it, and that will be useful as we examine the behavior of Android's task system.

A Canary for the Task's Coal Mine

In order to see some of the effects of fussing with our tasks, we need an app where we can see when our saved instance state comes and goes. To that end, we have the `Tasks/TaskCanary` sample application. It consists of a single activity, with a UI that is merely a full-screen `EditText`. In addition to the automatic saving of the `EditText` contents in the saved instance state `Bundle`, we also keep track of the time we first worked with that `Bundle`, in a data member named `creationTime`, backed by a `STATE_CREATION_TIME` entry in the `Bundle` itself:

```
package com.commonsware.android.task.canary;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.Settings;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends Activity {
  private static final String STATE_CREATION_TIME="creationTime";
  private long creationTime=-1L;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.activity_main);
  }

  @Override
  protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);

    dumpBundleToLog("restore", savedInstanceState);
    creationTime=savedInstanceState.getLong(STATE_CREATION_TIME, -1L);
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putLong(STATE_CREATION_TIME, getCreationTime());
    dumpBundleToLog("save", outState);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.settings) {
      startActivity(new Intent(Settings.ACTION_DATE_SETTINGS));
    }
```

```
    else if (item.getItemId()==R.id.other) {
      startActivity(new Intent(this, OtherActivity.class));
    }

    return(super.onOptionsItemSelected(item));
  }

  private long getCreationTime() {
    if (creationTime==-1L) {
      creationTime=System.currentTimeMillis();
    }

    return(creationTime);
  }

  // inspired by http://stackoverflow.com/a/14948713/115145

  private void dumpBundleToLog(String msg, Bundle b) {
    Log.d(getClass().getSimpleName(),
        String.format("Task ID #%d", getTaskId()));

    for (String key: b.keySet()) {
      Log.d(getClass().getSimpleName(),
          String.format("(%s) %s: %s", msg, key, b.get(key)));
    }
  }
}
```

(from Tasks/TaskCanary/app/src/main/java/com/commonsware/android/task/canary/MainActivity.java)

Each time we save and restore the instance state, we dump the `Bundle` to Logcat, so we can see what is in that `Bundle`. We wind up with lines like:

```
D/MainActivity: (save) android:viewHierarchyState: Bundle[...]
D/MainActivity: (save) creationTime: 1427032894794
D/MainActivity: (restore) android:viewHierarchyState: Bundle[...]
D/MainActivity: (restore) creationTime: 1427032894794
```

(where the `...` is a bit long to reproduce in the book and is not essential for the sample)

This way, both in the UI and in the logs, we can confirm that our state is being saved and restored as expected... or perhaps not as expected, in some cases.

You will notice that we have a pair of action bar items. One will bring up a screen from the Settings app, which we will use to see how this affects our task. The other

one will bring up another activity from our app, which we will use to explore how to start a clean task.

The Default User Experience

With all that behind us, let's start talking about tasks, focusing first on what behavior the developer gets "out of the box", with no task-specific logic in the app. In other words, what is the default user experience for an ordinary Android app?

NOTE: if you wish to reproduce the results described here, you will want to have Nukesalot and the Task Canary installed on your device or emulator.

Starting from the Home Screen

Assume that we are "starting from scratch". For example, the user has installed your app (or bought a device with your app pre-installed) but has never run your app before. Or, perhaps the overview screen is cleared of all tasks.

If the user taps your home screen launcher icon, not only is a process forked to run your app, but a new task is created, and your app's task will appear in the overview screen.

(see! that wasn't so hard!)

To reproduce this behavior:

- Clear the overview screen of all tasks, by swiping them off the screen. Note that this may take some time on an Android 5.x device that is really being used (versus just being some test device), as there may be a *lot* of tasks to clear.
- Run your app, from the home screen or IDE.

Resuming from the Overview Screen

Eventually, the user wanders away from your app. Then, later on, the user returns to your app, by finding the task associated with your app in the overview screen and tapping upon it.

In the end, you wind up in the same state as before: you have a process for your app, and your task is still in the overview screen. How we get there depends a bit on what

happened with your process, in between when you had been in the foreground and when the user taps on your task in the overview screen.

If your app's process was still running, nothing much happens of note, other than you return to the foreground. From a state standpoint, your app would be called with `onSaveInstanceState()` when the user left your app, but you will not be called with `onRestoreInstanceState()`, because your activity was not destroyed yet. Note that this assumes that you did not undergo a configuration change (e.g., user originally was in your app in portrait, then returned to you from the overview screen while the device was in landscape). In the case of a configuration change, your activity would be destroyed and recreated by default, and you *would* be called with `onRestoreInstanceState()`, but that would be due to the configuration change more so than the use of the task and the overview screen.

To reproduce the above behavior, given that your device was in the state after the "Starting from the Home Screen" section above:

- Press HOME to move your app to the background, and notice the "(saved)" entries being reported to Logcat.
- Quickly press RECENTS (or, if you have no such option, long-press HOME) to bring up the overview, and tap on your task there.

However, it is entirely possible that while your task is around that your process is terminated to free up memory for other processes. If the user returns to your app via the overview screen, a fresh process will be forked for your app. This would trigger a call to `onRestoreInstanceState()`, because your old activity no longer exists, because its process no longer exists.

To reproduce the above behavior, given that your device was in the state after the "Starting from the Home Screen" section above:

- Press HOME to move your app to the background, and notice the "(saved)" entries being reported to Logcat.
- Run Nukesalot, find the Task Canary in the list of running apps, and tap upon that entry to terminate its process. You should see its process go away in the list of debuggable processes in DDMS. You could also experiment with just terminating the process directly from DDMS, but Nukesalot may be a bit closer to "natural" device behavior, in terms of how the process is terminated.
- Press RECENTS (or, if you have no such option, long-press HOME) to bring up the overview, and tap on your task there. Note your "(restored)" Logcat

entries, which should include the same `creationTime` as you saved, even though it is now in the future.

Note that if you leave a task for an extended period of time — say, 30 minutes or so — the task may be "cleared" when you return to it. This means that you are taken back to whatever the "root" activity of the task is, where by "root" we mean the original activity put into the task.

Starting Another App

Some apps only start up other activities within the same app. However, many apps start up activities from other apps, either directly via `startActivity()` or indirectly (e.g., clicking in links in a `WebView`). For example, the Task Canary app has an item in the action bar overflow that, when clicked, brings up the Settings screen for adjusting date and time settings.

You might think that when the user taps on this overflow item, and Task Canary calls `startActivity()`, that a new task is created. After all, the Settings app is a completely separate app from the Task Canary app.

However, try this:

- Clear the overview screen
- Launch Task Canary
- Choose the Settings action bar overflow item to bring up the date-and-time Settings screen
- Press HOME to bring up the home screen
- Press RECENTS or otherwise bring up the overview screen

You will see one entry in the overview screen, for Task Canary, rather than two. Furthermore, particularly on Android 5.x devices, you will see the Settings screen as the top-most activity within the Task Canary task:

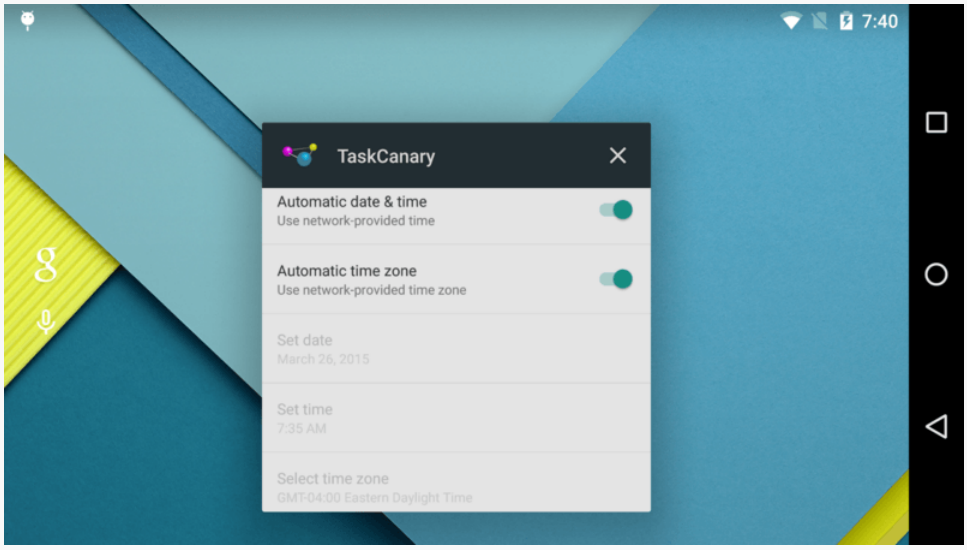


Figure 688: The Task Canary Task, on Android 5.1

However, suppose that instead of using `ACTION_DATE_SETTINGS` for the `Intent`, we used `ACTION_APN_SETTINGS` instead, to allow the user to view mobile access point names and such. You might think, given the above flow, that we would wind up with just one task, as we did with `ACTION_DATE_SETTINGS`. In reality, you will see two tasks, instead of just one:

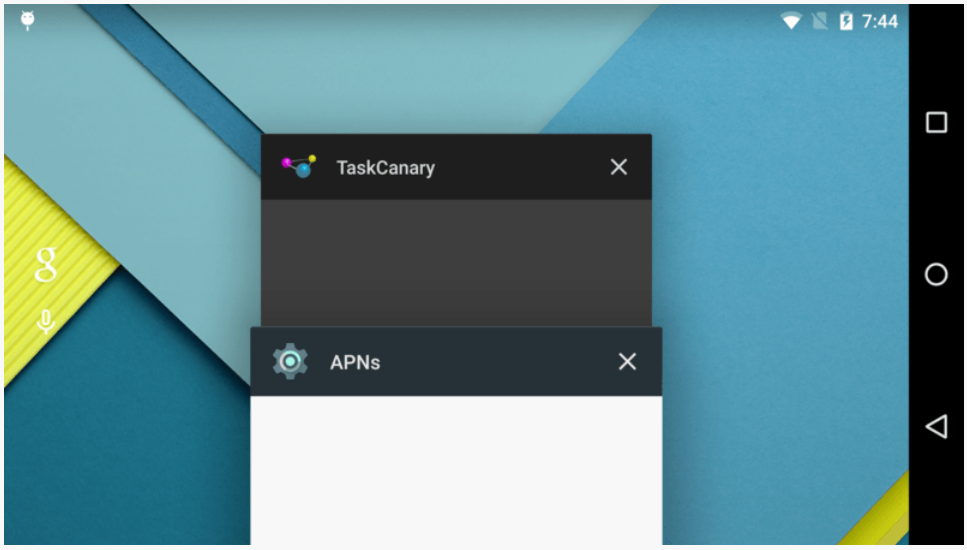


Figure 689: Two Tasks on Android 5.1

This is where things start to get a bit confusing.

Explaining the Default Behavior

With the user experience as background, let's now dive into what is really going on with these operations.

When Tasks are Created

A task is not created just because an activity is started. Otherwise, even individual apps would have lots of tasks, one per activity.

A task is not created just because a task from a different app is started. Otherwise, the two Settings scenarios above would have both resulted in a new task.

Instead, tasks are created when somebody *asks* for a task to be created. That "somebody" could be the author of the app calling `startActivity()` or the author of the activity being started.

There are three major approaches for indicating that a new task should be started: flags on the `Intent` used with `startActivity()`, task affinity values, and launch modes. We will get into launch modes later in this chapter, as the normally-used launch modes have no impact on tasks. Instead, we will focus on the other two approaches here.

Task-Management Intent Flags

If you want to start an activity, and ensure that the activity starts in a new task, add `Intent.FLAG_ACTIVITY_NEW_TASK` to the flags on the `Intent` being used with `startActivity()`:

```
startActivity(new Intent(SOME_ACTION_STRING)
                .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK))
```

What will happen, when you call `startActivity()` with `FLAG_ACTIVITY_NEW_TASK`, is that Android will see if there is a task that already has this activity in it. If there is, that task will be brought to the foreground, and the user will see whatever is on the top of that task's stack. Otherwise, if there is no task with this activity in it, Android will create a new task and associate a new instance of the activity with this task.

This is what home screen launchers do. When you tap on a home screen launcher icon, if there is a task that has a copy of your home screen activity in it, that task is brought back to the foreground. Otherwise, a new task is started.

If you *also* add `Intent.FLAG_ACTIVITY_MULTIPLE_TASK`, then Android skips the search for existing tasks and unconditionally launches the activity into a new task. This is generally not a good idea, as the user can wind up with many copies of this activity, not know which one is which, and perhaps have difficulty getting back to the right one.

Task Affinities

By default, if Android needs to create a new task as a result of `FLAG_ACTIVITY_NEW_TASK`, it just creates a task. And, if there is no such flag on the `Intent`, Android will put the activity into the task of whoever called `startActivity()`.

If, however, the activity has an `android:taskAffinity` attribute in its `<activity>` element in the manifest, then Android will specifically start this activity in a certain task, identified by the string value of the attribute. Other activities with the same task affinity will also go into this task.

The reason why the two Settings screens behave differently is that the `ACTION_APN_SETTINGS` activity has a certain task affinity value, while `ACTION_DATE_SETTINGS` does not. The task affinity of the `ACTION_APN_SETTINGS` activity is shared by many, though not all, activities within the Settings app. Those activities, when started, will always go into the task identified by the affinity. Hence, when we start `ACTION_DATE_SETTINGS`, it goes in *our* task (because that activity has no affinity and we did not include `FLAG_ACTIVITY_NEW_TASK`), but when we start the `ACTION_APN_SETTINGS` activity, it goes into a Settings-specific task.

Note that you can also have the `android:taskAffinity` value defined on the `<application>` element, to provide a default task affinity for all activities. The overall default is `""`, or no affinity.

When Tasks are Removed

On Android 3.0 and higher, the user can get rid of a task by swiping the task off of the overview screen.

Otherwise, prior to Android 5.0, a task would automatically go away after some amount of user activity, as there were only so many "slots" available for tasks.

On Android 5.0+, though, it is unclear if there is an upper bound to how many tasks can exist. Beyond that, tasks survive a reboot, as information about those tasks is persisted. We will get more into the ramifications of this, and how you can take advantage of it, later in this chapter.

When Tasks (and Processes) are Resumed

A task will be resumed and brought back to the foreground in several situations, including:

- the user manually requests it via the overview screen, by clicking on one of the recent tasks
- if `FLAG_ACTIVITY_NEW_TASK` is added (without `FLAG_ACTIVITY_MULTIPLE_TASK`) to the `Intent` used to start an activity, and there is a task containing the activity in question
- if the `taskAffinity` for the activity being started ties it to another task
- if the launch mode for the activity being started ties it to another task

However, just because the task exists does not mean that the process(es) exist for the activities in the task. As needed, Android will fork fresh processes, to be able to load in the app's code and start the necessary activities. Android will deliver to the newly-created activities the same `Intent` that was used to create the original incarnation of the activity (via `getIntent()`) and the saved instance state `Bundle`.

What Happens to Services

In theory, services are immune to task behavior. Tasks can come and go, and services are usually oblivious to this.

A service should be called with `onTaskStopped()` if a task associated with one or more of the app's activities is removed. The service might use that as a signal that it too should shut itself down.

There appears to be a quasi-documented `android:stopWithTask` attribute on the `<service>` element in the manifest. The default is `false`, but if you override it to be `true` on your `<service>`, then `onTaskStopped()` will not be called, and Android will simply destroy your service when the task is removed.

However, as of Android 4.4, there are many reports that services may be destroyed when a task is removed, even without `android:stopWithTask="true"`, though on a slight delay. Developers concerned about this should keep an eye on this issue and this issue, both for various hacky workarounds and for any signs that this is being permanently addressed.

What's Up with onDestroy()?

If the user swipes away the task using the overview screen, `onDestroy()` will be called on all outstanding activities.

If, however, you use Nukesalot to kill the background process, Android does *not* call `onDestroy()` on any outstanding activities. Since other task killers will use the same techniques as does Nukealot, this means that your `onDestroy()` methods will not be called when your process is terminated by those apps as well.

So, removing a *task* is a graceful exit, and Android calls `onDestroy()`, but an explicit termination of your process by another is a not-so-graceful exit, and Android skips `onDestroy()`.

As a result, as previously advised in this book and elsewhere, you cannot count on your `onDestroy()` methods being called, and you need to take this into account in terms of what sorts of code you put in them.

Basic Scenarios for Changing the Behavior

In many cases, the default behavior of tasks is just fine. However, there are many scenarios in which we may want to override the default behavior, routing activities to specific tasks, to have a better flow for the user.

Reusing an Activity

By default, each time you call `startActivity()`, a new instance of the activity is created. Depending upon the user flow, that may not be a bad approach. For example, it may be that the only logical path out of the started activity will be to press BACK and destroy it.

However, there will be plenty of cases where we will not want to keep creating new activity instances. For example, if you elect to have several activities reachable via a nav drawer, you do not want to create fresh instances of activities that the user has

already visited via that drawer. Otherwise, they will keep piling up, continuing to consume heap space. Instead, it would be better to try to reuse an existing activity instance, if one is available, creating a fresh one only if needed.

The most flexible approach for accomplishing this involves using a flag on the `Intent` used to start the activity: `Intent.FLAG_ACTIVITY_REORDER_TO_FRONT`. This tells Android to bring an existing activity matching our `Intent` to the foreground, if one already exists in our task. If there is no such activity, then go ahead and create a new instance.

The `Tasks/RoundRobin` sample application demonstrates this. It consists of two activities (`FirstActivity` and `SecondActivity`), each of whose UI consists of one really big button. Clicking the button should start the other activity, so clicking the button in `FirstActivity` should start an instance of `SecondActivity`. But, we want to reuse activity instances where available, and confirm that indeed we *are* reusing those instances.

`FirstActivity` accomplishes that by adding `FLAG_ACTIVITY_REORDER_TO_FRONT` to the `Intent` used to start `SecondActivity` when the button is clicked:

```
package com.commonsware.android.tasks.roundrobin;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class FirstActivity extends Activity implements View.OnClickListener {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.first);
    findViewById(R.id.button).setOnClickListener(this);

    Log.d(getClass().getSimpleName(),
        String.format("onCreate for %x", hashCode()));
  }

  @Override
  protected void onResume() {
    super.onResume();

    Log.d(getClass().getSimpleName(),
```

```
          String.format("onResume for %x", hashCode()));
  }

  @Override
  protected void onDestroy() {
    Log.d(getClass().getSimpleName(),
        String.format("onDestroy for %x", hashCode()));

    super.onDestroy();
  }

  @Override
  public void onClick(View view) {
    startActivity(new Intent(this, SecondActivity.class)
                    .addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT));
  }
}
```

(from Tasks/RoundRobin/app/src/main/java/com/commonsware/android/tasks/roundrobin/FirstActivity.java)

`SecondActivity` has a nearly identical implementation, just routing back to `FirstActivity`.

If you run the app, the user's perspective is that clicking the button "ping-pongs" the user between the two activities. Looking at Logcat, you will see new instances created the first time the user visits an activity, courtesy of the `Log.d()` call in `onCreate()`. But, if the user returns to an existing instance via the button click, you will see that `onCreate()` is not called, and that the `hashCode()` reported in `onResume()` matches the `hashCode()` of the previously-created instance of this activity:

```
D/FirstActivity: onCreate for b31b9430
D/FirstActivity: onResume for b31b9430
D/SecondActivity: onCreate for b31eb8a8
D/SecondActivity: onResume for b31eb8a8
D/FirstActivity: onResume for b31b9430
D/SecondActivity: onResume for b31eb8a8
```

If you use Nukesalot to terminate the process for RoundRobin, then return to RoundRobin (e.g., via the overview screen), you will see that Android has to create a new instance of whatever activity had been in the foreground, as the old instance went away when the old process did. An instance of the other activity will not be created until the user returns to it, such as via a click of the really big button. In other words, Android lazy-instantiates the activities in the task's back stack, only creating instances when it is absolutely required based upon user navigation.

Note that launch modes offer another way to control this behavior, having the activity being started indicate that its instance should always be reused. However, this is a specialty case, one that most apps will not require.

Forcing a Clean Task

Let's suppose that you have an app that requires in-app authentication, via some form of login screen. For example, your app's data is held in SQLCipher for Android, and so you need the user to supply a passphrase for the database.

In the beginning, when your app is launched from the home screen, your `LAUNCHER` activity appears. If that is your login screen, all is good. You collect the passphrase, create your singleton instance of the SQLCipher-enabled `SQLiteOpenHelper`, and you can access the database.

Eventually, the user presses HOME, and time passes. Android terminates your process to free up system RAM. The user then tries returning to your existing task, such as via the overview screen. Android creates a fresh process for you and takes you to the activity on the top of that task's back stack. But at this point, your singleton `SQLiteOpenHelper` is gone, and you need to collect a passphrase again.

You might think that this is purely a UI issue. Rather than collecting the passphrase in an activity, you collect it in a fragment, one that your `LAUNCHER` activity uses directly, and one that other activities can use via a `DialogFragment`. This way, you can arrange for every activity to be able to complete the re-initialization of your process and give you access to the encrypted database again.

Another approach would be to say that you want to wipe out this task and start over, routing the user back to the `LAUNCHER` activity for authentication.

There are two main approaches for implementing this: setting `Intent` flags or using `android:clearTaskOnLaunch` in the manifest.

Starting a Cleared Task Yourself

One way to do that is to have each activity check to see if a new task is needed (e.g., "is the `SQLiteOpenHelper` singleton `null`?"). When that situation is detected, you call `startActivity()` for your `LAUNCHER` activity, with two flags: `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_CLEAR_TASK`.

For example, the Tasks/Tasksalot sample application is a straight-up clone of Launchalot with only one change of substance: using FLAG_ACTIVITY_CLEAR_TASK instead of FLAG_ACTIVITY_RESET_TASK_IF_NEEDED:

```
@Override
protected void onListItemClick(ListView l, View v,
                               int position, long id) {
  ResolveInfo launchable=adapter.getItem(position);
  ActivityInfo activity=launchable.activityInfo;
  ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                       activity.name);
  Intent i=new Intent(Intent.ACTION_MAIN);

  i.addCategory(Intent.CATEGORY_LAUNCHER);
  i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
             Intent.FLAG_ACTIVITY_CLEAR_TASK);
  i.setComponent(name);

  startActivity(i);
}
```

(from Tasks/Tasksalot/app/src/main/java/com/commonsware/android/tasksalot/MainActivity.java)

To see this in action:

- Run the TaskCanary sample app and use the overflow to bring up OtherActivity
- Press HOME
- Run Tasksalot
- Click on the "Task Canary" entry in Tasksalot

At this point, you will see the TaskCanary sample app return to the screen. From the logs in Logcat, you will see it is the same task ID as before. Yet, you are seeing the FirstActivity. OtherActivity was removed from the task as part of FLAG_ACTIVITY_CLEAR_TASK processing.

This differs from what you see in a home screen, with FLAG_ACTIVITY_RESET_TASK_IF_NEEDED. If you run the same test, but rather than use Tasksalot, you tap on the "Task Canary" icon in the home screen launcher, the task will return to the foreground, but you will be taken to OtherActivity. FLAG_ACTIVITY_CLEAR_TASK *always* clears the task and makes the activity that you are starting up be the root of the newly-cleared task.

Always Starting a Cleared Task

Perhaps you *always* want to start with a cleared task, whenever the user returns to the task after having left it previously. In other words, you always want to start back at whatever your task's root activity is, which is typically your launcher activity.

To do this, simply have `android:clearTaskOnLaunch="true"` on that launcher activity. Then, for any task where that activity is the root, when the user returns to the task, any other activities in the task are reparented (if applicable) or dropped.

Note, though, that this does not mean that you get a new *process*. Hence, any singletons you had before may or may not still be there.

So, in the authentication scenario described above, using `android:clearTaskOnLaunch="true"` would take the user back to your initial activity, where you can perform the authentication. However, if you detect that the `SQLiteOpenHelper` still exists, and therefore you do not need the user to log in again, you could switch over to showing your initial content (e.g., run a `FragmentTransaction`).

This is far simpler than having the detect-the-`null`-singleton-on-each-activity approach. However, the downside is that the user loses context. If they were six activities deep into your app, and they get interrupted by a phone call, when they come back to your app, they are back at the beginning.

Launching an App Into a New Task

A home screen launcher app, when it invokes the user's selected activity, will use code something like this from the `Launchalot` sample:

```
@Override
protected void onListItemClick(ListView l, View v,
                               int position, long id) {
  ResolveInfo launchable=adapter.getItem(position);
  ActivityInfo activity=launchable.activityInfo;
  ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                       activity.name);
  Intent i=new Intent(Intent.ACTION_MAIN);

  i.addCategory(Intent.CATEGORY_LAUNCHER);
  i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
             Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);
  i.setComponent(name);
```

```
    startActivity(i);
  }
```

(from Introspection/Launchalot/app/src/main/java/com/commonsware/android/launchalot/Launchalot.java)

Here, we:

- Create a `ComponentName` identifying the specific activity in the specific app to be started (in this case, based on the `ResolveInfo` that the user chose)
- Create an `Intent` for the `MAIN` action and the `LAUNCHER` category
- Set the `FLAG_ACTIVITY_NEW_TASK` and the `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` flags in the `Intent`
- Attach the `ComponentName` to the `Intent`, to convert it from an implicit `Intent` into an explicit `Intent`
- Start the activity using the `Intent`

`FLAG_ACTIVITY_NEW_TASK` indicates that we want the activity being started to be the root of a new task. If there is no outstanding task for this app, a new task will be created, a new activity instance will be created, and that activity will be the root of the task. Here, "root" means that if the user presses BACK and destroys the activity, the task itself is removed and the user returns to the home screen.

However, despite its name, `FLAG_ACTIVITY_NEW_TASK` does not necessarily create a new task. If there is an existing task for this app containing this activity, that task is brought back to the foreground and is left intact. The activity we request is not created, let alone brought to the foreground.

That is where `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` comes in. It ensures that the task that is brought to the foreground is showing the requested activity. This may involve reparenting activities as well.

Another possibility, instead of `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED`, is `FLAG_ACTIVITY_MULTIPLE_TASK`. This *always* starts a fresh task, with a fresh instance of the requested activity in the root of that task. However, this now may mean that the user has multiple tasks for the same app, which may be confusing in some circumstances. However, this also lies at the core of Android 5.0's documents-as-tasks support and therefore may become more familiar to users over time.

The Invisible Activity

Several sample apps in this book use an "invisible activity", one with the theme set to `Theme.Translucent.NoTitleBar`. These are useful in cases where something outside your app needs an activity, but you do not really have a UI that you want to display. In the case of the book samples, having a `LAUNCHER` activity makes it much easier for readers like you to simply run the samples from the IDE.

However, those sample apps usually do not have any other activities. The invisible activity is just there to kick-start something else, such as `AlarmManager` events.

However, if you have a mix of invisible and regular activities in an app, your invisible activities still wind up potentially having a visible impact.

For example, suppose that we have an ordinary Android app, with regular activities. However, we want a home screen shortcut icon to allow the user to start something in the background, such as playing music. While an app widget would allow us to control what happens when the user taps on an icon in that app widget, a home screen shortcut icon *always* launches an activity. So, we make the start-the-music activity invisible via `Theme.Translucent.NoTitleBar`.

If the user taps on that shortcut, and none of our other activities are part of a task, things proceed as expected: the music starts and the user sees nothing (other than perhaps a `Toast` that we show to let the user know that we are responding to their request).

But, if one or more of our activities are in some task, launching the invisible activity *brings the task back to the foreground*. While our invisible activity is still invisible, the user now sees whatever other activity of ours they had last been in. It is possible that this is a feature, and not a bug, for some apps. But, in other cases, we might want the invisible activity to not have this effect.

The solution: task affinity.

Your ordinary activities can use the default task affinity, or have other task affinities as needs dictate. Your invisible activity, though, would have an `android:taskAffinity` value that is distinct from all others, to force it into its own task. That way, when the user taps on the shortcut, the invisible activity routes to its own task. That task will not yet exist, so the invisible activity causes the task to be created. When the invisible activity calls `finish()` to destroy itself after kicking off the background work, the task is now empty and is removed. Since this was a new

task, no existing UI would be brought back to the foreground, and since the task is removed in the end, we are "reset" for the next time the user taps on the shortcut.

Reparenting Tasks

One of the more unusual features of Android's task system is the ability for activities to be "reparented", or moved from one task to another. On the surface, this feels a bit odd, as if a Web page on one browser tab might magically show up in a separate browser tab, just via navigation. And, in truth, it is a specialized use case, but one that could conceivably apply to your app.

Suppose that you were writing an SMS client. You have an activity that is your message composer, where the user can type in a text message to send to somebody. You export that activity, with `Intent` actions like `ACTION_SEND` and `ACTION_SENDTO`. A third-party app, using one of those `Intent` actions, starts up your message composer activity. In the absence of a `taskAffinity` to stipulate otherwise, by default, your message composer activity will be in the task of the third-party app.

Now, suppose that the user fails to actually send a message, such as by pressing HOME from the third-party app's task. Some time later, the user taps on your app's home screen launcher icon. At this point, there are two possibilities as to what happens:

1. You may decide that you want to have the already-running message composer activity appear, to remind the user that they were in the middle of composing a text message and failed to either send it or explicitly BACK out of the activity.
2. You may decide that you do not care, and you are willing to ignore that outstanding message composer activity instance.

The default is option #2. If, instead, you want to offer option #1, that is where task reparenting comes into play.

On your `<activity>` (or on `<application>` to set an app-wide default), you can have `android:allowTaskReparenting="true"`. This indicates to Android that the message composing activity, that is on some other app's task, can move to your app's task when that task is created.

The trigger for this "reparenting" is the task affinity. If you do not specify a task affinity for an activity, the default affinity is for a task rooted in one of your app's activities, typically the launcher activity. In some circumstances, when a task for

your app is created, Android will search through other tasks to see if there is any activity, in another task, that has an affinity for your task and allows reparenting. If there is a match, that activity is brought into your task.

The "some circumstances" mentioned in the preceding paragraph is something using two `Intent` flags when calling `startActivity()`:

- `FLAG_ACTIVITY_NEW_TASK`, to create a new task if one is needed, and
- `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED`, to clear out the task if it already has contents *and* reparent any activities in other tasks to this one if appropriate

As it turns out, home screen launchers are supposed to use this pair of flags when they respond to the user tapping on a home screen launcher icon.

The `Tasks/ReparentDemo` sample Android Studio project contains a pair of applications as modules that demonstrate this effect, based on David Wasser's epic Stack Overflow answer.

One module, `app/`, contains an application with two activities, where the second activity (`ReparentableActivity`) has `android:allowTaskReparenting="true"`:

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.tasks.reparent">

  <application
    android:allowBackup="true"
    android:label="@string/app_name"
    android:icon="@drawable/ic_launcher">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity
      android:name=".ReparentableActivity"
      android:allowTaskReparenting="true">
      <intent-filter>
        <action android:name="com.commonsware.android.tasks.reparent.WHEEEEE"/>
        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

(from Tasks/ReparentDemo/app/src/main/AndroidManifest.xml)

The two activities just display static messages, indicating which of those two activities you are seeing in the foreground. They also log process and task IDs to Logcat. `MainActivity` does that in `onCreate()`:

```
package com.commonsware.android.tasks.reparent;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    Log.d(getApplicationInfo().loadLabel(getPackageManager()).toString(),
          String.format("Process ID %d, Task ID %d",
              android.os.Process.myPid(), getTaskId()));
  }
}
```

(from Tasks/ReparentDemo/app/src/main/java/com/commonsware/android/tasks/reparent/MainActivity.java)

`ReparentableActivity` logs the same information in `onResume()`:

```
package com.commonsware.android.tasks.reparent;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class ReparentableActivity extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.reparent);
  }

  @Override
  public void onResume() {
    super.onResume();

    Log.d(getClass().getSimpleName(),
        String.format("Process ID %d, Task ID %d",
```

```
            android.os.Process.myPid(), getTaskId()));
  }
}
```

(from Tasks/ReparentDemo/app/src/main/java/com/commonsware/android/tasks/reparent/ReparentableActivity.java)

The other module, `app2/`, contains an application with one activity, whose UI consists of one really big button. Clicking that button triggers a `launch()` method that calls `startActivity()` on an `Intent` identifying the `ReparentableActivity` from the first app:

```
package com.commonsware.android.tasks.reparent.app2;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    Log.d(getApplicationInfo().loadLabel(getPackageManager()).toString(),
        String.format("Process ID %d, Task ID %d",
            android.os.Process.myPid(), getTaskId()));
  }

  public void launch(View v) {
    startActivity(new Intent("com.commonsware.android.tasks.reparent.WHEEEEE"));
  }
}
```

(from Tasks/ReparentDemo/app2/src/main/java/com/commonsware/android/tasks/reparent/app2/MainActivity.java)

To see this behavior in action, install both apps. If you run them straight from your IDE, you will want to clear out all relevant tasks, either by swiping them off the recent-tasks list (or by rebooting the device or emulator, if it runs Android 4.4 or lower).

Then, start up the "Reparent Demo Aux" app (from the `app2/` module). Click the button, and you will see the `ReparentableActivity` appear. If you press HOME, bring up the recent-tasks list, and go back to this task, you will see the same `ReparentableActivity`. The task, however, is for "Reparent Demo Aux".

Now, press HOME, then start up the "Reparent Demo" app (from the `app/` module). Rather than seeing the `MainActivity` from that app, you see the `ReparentableActivity` instance from before. The logs will illustrate that your

process ID has not changed, but that the task ID for this activity *has* changed, from the task ID used by the `app2/` app to the task ID created for `app/`. The activity has been reparented.

The use of `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` may sound a lot like `FLAG_ACTIVITY_CLEAR_TASK`. The "if needed" part comes into play in two cases:

- If a new task is being created, the "reset" work is really the reparenting described above
- If an existing task is being brought back to the foreground, then get rid of resettable activities

Here, by "resettable activities", we mean:

- Activities launched with the `FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET` flag
- Any activities that are higher on the back stack than other explicitly resettable activities

So, if our back stack consists of activities A-B-C-D, and C was started with `FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET`, and we start up one of these activities (say, A) with `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED`, and this existing task is coming back to the foreground, C and D will be cleared from the task. The user ordinarily would be taken to activity D, but instead will be taken to activity B, because C is explicitly resettable and D is higher on the back stack.

The Self-Destructing Activity

Sometimes, you only want an activity around while it is in the foreground and the user can see it. Once the user leaves the app, you no longer want that activity to exist. For example, a bank app showing bank account details might want this behavior, so that highly-sensitive information like this does not hang around. Or, you might want this for certain activities that are memory-intensive, so they release their heap space and reduce the odds of an `OutOfMemoryError`.

You could attempt to manage this yourself, via timely calls to `finish()`, but catching all the cases when `finish()` is needed could get troublesome.

Instead, Android has a pair of options to have no-history activities: activities that automatically finish when the user leaves them:

- An activity can decide for itself that it should be removed upon a task switch via the `android:noHistory` attribute on the `<activity>` in the manifest
- You can decide ad-hoc to have activities exhibit this behavior by adding `Intent.FLAG_ACTIVITY_NO_HISTORY` on the `Intent` used to start those activities

You can see these in action in the `Tasks/NoHistory` sample application. This is a near-clone of a simple two-activity app that we saw back when we first learned about how to have multiple activities.

There are only two real differences in this version of the sample app.

First, the launcher activity (`MainActivity`) has `android:noHistory="true"` on its `<activity>` element:

```
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name"
      android:noHistory="true">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```

(from Tasks/NoHistory/app/src/main/AndroidManifest.xml)

Second, when that activity goes to start `OtherActivity`, it adds `FLAG_ACTIVITY_NO_HISTORY` to the `Intent` used with `startActivity()`:

```
public void showOther(View v) {
  Intent other=new Intent(this, OtherActivity.class);

  other.putExtra(OtherActivity.EXTRA_MESSAGE,
                 getString(R.string.other));
  other.addFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);

  startActivity(other);
}
```

(from Tasks/NoHistory/app/src/main/java/com/commonsware/android/tasks/nohistory/MainActivity.java)

Both of these inherit from the original sample's `LifecycleLoggingActivity`, which just logs messages to Logcat on the major lifecycle methods. If you run the app, click

the big button to go from `MainActivity` to `OtherActivity`, then switch to some other app (via the overview screen, via the home screen launcher, etc.), you will see that both activities are destroyed, even though we do not press BACK, call `finish()`, or do anything else ourselves to destroy them.

This has a key side-effect: you cannot combine no-history with `startActivityForResult()` especially well. If the activity that calls `startActivityForResult()` has no-history enabled (via the manifest attribute or the `Intent` flag), it will simply not be called with `onActivityResult()`.

A related attribute is `android:finishOnTaskLaunch`. If set to `true`, and if the user leaves the task and returns to it, the activity is destroyed. Whereas `android:noHistory` removes the activity when the user leaves the *activity*, `android:finishOnTaskLaunch` only removes the activity when the user leaves the task and returns to it.

The Hidden Task

Perhaps you have a use case where you want your entire task to be hidden from the overview screen.

To do that, you can indicate that the activity that is the root of the task (e.g., your launcher activity) is to be "excluded from recents". To do that, you can:

- Add `android:excludeFromRecents="true"` to the appropriate `<activity>` element in your manifest, or
- Add `Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS` to the `Intent` used to start up the activity and its task

Note that this only matters if the activity in question is the task root (i.e., the one that started the task). Having this setting on other activities higher in the back stack will have no effect on the visibility of the task.

Also, please note that this does not eliminate the task itself. It merely hides it from the overview screen. So, for example, suppose you were to:

- Add `android:excludeFromRecents="true"` to `MainActivity` in the `TaskCanary` sample
- Run the sample app
- Press HOME and note the task ID that shows up in Logcat
- Press RECENTS and note that the task does not show up there

- Return to the home screen, find `TaskCanary` in the launcher, and tap on the launcher icon
- Press HOME again and note the task ID that shows up in Logcat

You will see that those task IDs are the same. So, the task is there, and we can return to that task, but the task is merely suppressed from the listing shown in the overview screen.

Dealing with the Persistent Tasks

As noted previously in this chapter, on Android 5.0+, tasks live forever, insofar as they survive a reboot. That, coupled with a seemingly-infinite roster of recent tasks — compared with rather finite lists in earlier versions of Android — means that your app usually will be brought back from an existing task on Android 5.0+.

However, there are a few key differences.

The State of Your State

For normal process termination, in between device reboots, the `Bundle` that we get in `onSaveInstanceState()` is held onto in RAM by some core OS process. Of course, on a reboot, that process is terminated along with everything else. And a `Bundle` can hold onto objects that, while perhaps `Parcelable`, are not designed to be persisted.

The default behavior is that when your task is brought back to the foreground after a reboot, only the task's root activity is created, and that is the only activity in the task. This effectively mimics the behavior of pre-Android 5.0 versions of Android.

However, if you want to, you can control a bit more how your task behaves on a reboot.

Your `<activity>` element in the manifest can have a largely-undocumented `android:persistableMode` attribute. If you set this to `persistAcrossReboots` on the activity that serves as the root of your task (e.g., your launcher activity), then you will be able to override three additional methods on your `Activity`:

- `onCreate()`
- `onSaveInstanceState()`
- `onRestoreInstanceState()`

Right now, you may think that the author of this book is drunk, as we covered those methods already, far earlier in the book.

However, what API Level 21 adds, for `persistAcrossReboots` activities, are flavors of those methods that take two parameters: a `Bundle` (as normal) and a `PersistableBundle`. Values that you store in the latter parameter will be delivered to you when your activity is re-created as part of your task coming back to the foreground, even after a reboot.

Note that all of the above requires that you set your `compileSdkVersion` to 21 or higher.

`PersistableBundle` allows you to save `int`, `long`, `double`, and `String` values, along with arrays of each. On Android 5.1+, you can also save `boolean` and arrays of `boolean` values. Notably, you cannot put a `Parcelable` (or, strangely, a `Serializable`) in a `PersistableBundle`.

If your activity has `persistAcrossReboots` set — as does `MainActivity` in the `Tasks/PersistentCanary` sample application — you will be called both with the single-parameter and dual-parameter versions of those methods, in that order. Unless your app has a `minSdkVersion` of 21 or higher, you will probably wind up overriding both versions of each method, where you put stuff in the `Bundle` in the single-parameter method and you put stuff in the `PersistableBundle` in the dual-parameter method. Since versions of Android prior to 5.0 do not know about `PersistableBundle` or methods that take one, only the single-parameter versions of those methods will be called on those devices. If your `minSdkVersion` is 21 or higher, though, you could just override the dual-parameter versions of the methods and work with both `Bundle` and `PersistableBundle` as needed.

Where You Return To

Normally, if your task is in the overview screen, and the user returns to it, the user will be taken to whatever activity was at the top of the back stack.

However, if the device reboots, and the user returns to your task, what happens depends on that semi-documented `persistableMode` value:

- If the value for the root activity of the task is `persistNever`, the task is not persisted across reboots

- If the value for the root activity of the task is `persistRootOnly`, the task will be persisted, but only for that root activity; other activities higher on the back stack are discarded
- If the value for the root activity of the task is `persistAcrossReboots`, then not only is the task persisted for the root activity, but other activities on the back stack are also persisted if they too have `persistAcrossReboots` (and were not launched with the `FLAG_CLEAR_TASK_WHEN_RESET` flag)

So, in the case of `PersistentCanary`, even if you use the overflow to bring up the date-and-time Settings screen, since that activity has the default `persistRootOnly` value for `persistableMode`, only the `MainActivity` will be in the task after a reboot.

Documents As Tasks

Tasks used to be relatively app-centric. By and large, each app had its own task, and just one task.

Android 5.0 extended the task system to support the notion of "documents" as tasks. Now, an app may be in several tasks, with different tasks focused on different "documents" or other specific contexts.

The vision is that this would be used by:

- Web browsers, where different browser tabs would be represented as separate tasks
- Editors, where different editing sessions on different content could be represented as separate tasks
- And so on

The benefit to the user is a standard way to switch between these different contexts, by means of the overview screen. The risk is that the overview screen becomes unwieldy, choked with too many entries to sift through.

When You Should Do This

An app should open a new "document" based on some specific explicit "open" operation by the user. So, for example:

- If the user asks to open a new "tab" in a browser, that could start a new document

- If the user asks to open a new file into an editor, that *might* start a new document, if you feel that the user understands that there are N other documents out there already opened in this editor
- If the user launches one of your activities from outside of the home screen, such as by clicking on a link in a Web browser, that might start a new document, to keep that work separate from any past work that might be part of other active tasks

Conversely, an app should not open a new document based on pure navigation operations:

- Swiping to a new page in a `ViewPager` should not open a new document
- Choosing an item in a nav drawer should not open a new document
- Tapping on an action bar item on its own should not open a new document, though it might lead the user down a path to open a new document

Adding a Document

You have a few options for launching an activity as a new document, indicating that it should have a separate entry on the Android 5.0+ overview screen.

android:documentLaunchMode

If you always want this activity to form the basis of a new document, add `android:documentLaunchMode="always"` to the `<activity>` element of your manifest, and you are done. Every time you start up an instance, you will get a new document.

This can be seen in the `Tasks/Docs` sample application, which has an `EditorActivity` with the aforementioned attribute:

```
    <activity
      android:name=".EditorActivity"
      android:documentLaunchMode="always"
      android:maxRecents="3"
      android:autoRemoveFromRecents="true"/>
```

(from Tasks/Docs/app/src/main/AndroidManifest.xml)

(we will cover those other new attributes shortly)

There are four possible values for `android:documentLaunchMode`:

- always, as noted, always starts a new document
- intoExisting, which looks for an existing document, where the root activity is the same class and the Intent is for the same Uri, and brings it back to the foreground, or starts a new document if a match cannot be found
- never prevents this activity from ever being launched as a new document
- none, which is the default, indicates that the activity will only be launched as a new document if Intent flags indicate that it should, as will be explained shortly

Since intoExisting depends upon Uri matches, you only want to use intoExisting if you are passing Uri values into the activity when starting it. Otherwise, use always.

FLAG_ACTIVITY_NEW_DOCUMENT

To conditionally launch an activity as a new document, have its android:documentLaunchMode set to none (or missing, since that is the default), and add Intent.FLAG_ACTIVITY_NEW_DOCUMENT to the Intent that is used to start up the activity that would represent a new document. This will have the behavior akin to intoExisting for android:documentLaunchMode, meaning that Android will search for a matching document and bring it back to the foreground if the match is available.

To replicate always functionality, add both Intent.FLAG_ACTIVITY_NEW_DOCUMENT and Intent.FLAG_ACTIVITY_MULTIPLE_TASK to the Intent.

Capping the Number of Documents

By default, you can launch as many documents as you want. However, unless you get rid of the document (as will be described below), or the user gets rid of the document (by swiping it off the overview screen), your roster of documents can keep piling up. Users may get frustrated if their overview screen is flooded by entries for your app.

You can employ an automatic least-recently-used (LRU) algorithm here by adding android:maxRecents to the <activity> that is the root of the task for the document. This indicates the maximum number of entries there should be in the overview screen for that activity, where Android will remove older tasks to make way for new ones if needed.

So, in the Docs sample, android:maxRecents="3" limits the number of EditorActivity tasks to 3; if the user tries opening more than this, older ones are quietly removed.

Note that the default value for android:maxRecents is 16. Also, there is a cap, ranging from 25 to 50, depending on device RAM — you will be unable to set it higher than this.

Removing and Retaining Documents

Android's default behavior is that the document will exist forever, or until the user swipes it off the overview screen.

It is rather unlikely that this is really the behavior that you or your users will want. Hence, you are going to want to take some steps to ensure that your documents will go away from the overview screen when they are no longer needed.

The simplest solution is to add android:autoRemoveFromRecents="true". This indicates that once the root activity is finished (e.g., the user presses BACK), the document is removed. By default, pressing BACK does *not* remove the document, so you need to opt into this behavior.

However, that approach assumes that it is fairly easy for the user to get back to the task's root activity and press BACK. If you have a complex navigation of activities within the "document", it may not be easy for the user to trigger document removal this way.

You can also forcibly get rid of the document by calling finishAndRemoveTask() yourself on an activity in the task. For example, in a tabbed Web browser, if you have a "close tab" UI element (e.g., action bar item), that could call finishAndRemoveTask() to get rid of the "document".

Other Task-Related Activity Properties

There are other attributes that you can place on your <activity> element in the manifest that have impacts on how that activity participates with the task system.

launchMode

Occasionally, particular techniques become much too popular in Android development, courtesy of some blog posts or other resources touting them as "quick hacks" to address certain issues. The `android:launchMode` attribute is one of those. Most Android apps should have no need to change `launchMode` off of its default value of `standard`, or occasionally `singleTop`. Yet, because the Android task system is rather confusing, some developers latch onto other launch modes and use them in places where there are better, more fine-grained solutions.

That being said, let's explore the launch modes, with the help from the fine people at Novoda. The Novoda developers released an app on the Play Store, and an accompanying GitHub repo that helps to illustrate the launch modes.

That app has four activities, one for each of the four launch modes:

- `standard`
- `singleTop`
- `singleTask`
- `singleInstance`

The launcher activity is the `standard` activity. Each activity has four buttons, to start up that activity via `startActivity()`, by default with no particular `Intent` flags (though there's a legacy options menu that allows you to play with those as well). The color-coded UI for each activity also shows a unique identifier of the activity, the task ID of the task that the activity is in, the lifecycle methods that were invoked on that instance, and a set of stacked bars designed to illustrate what should be on the back stack for that task (using some techniques of dubious reliability, but the sort of thing that should be OK for a demo app like this).

So, when we launch the app, we get a green UI for a `standard` activity:

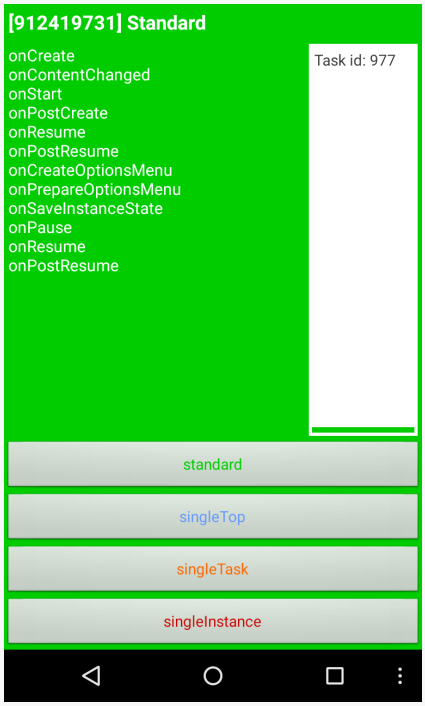


Figure 690: Novoda Demo App, As Initially Launched

`singleTop`

Using `singleTop` for the `launchMode` has one effect: controlling whether a new instance of the activity is created. Normally, calling `startActivity()` will create a new instance of the activity, unless `Intent` flags dictate otherwise. With `singleTop`, if the activity being started is already at the top of its stack, that existing instance is simply called with `onNewIntent()`. Otherwise, `singleTop` behaves as does `standard`.

So, if we tap the button to launch a `singleTop` method in the Novoda demo app, from our earlier state, we get a blue `singleTop` activity:

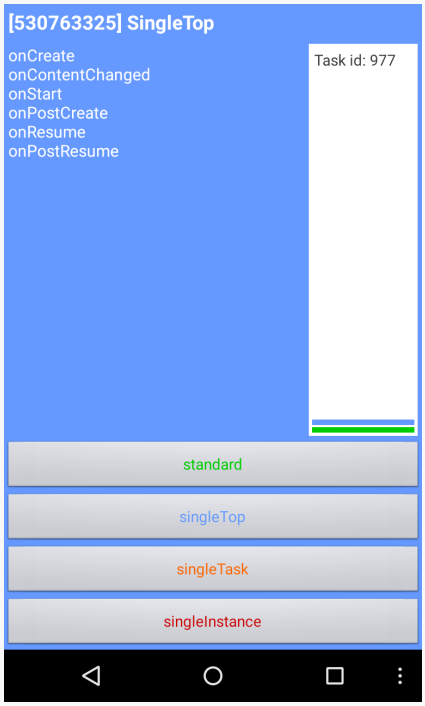


Figure 691: Novoda Demo App, After Starting singleTop Activity

That worked just like `standard`. But, if we tap the same button again, we do *not* get a new instance of the activity. However, the transcript of lifecycle methods shows that `onNewIntent()` was called:

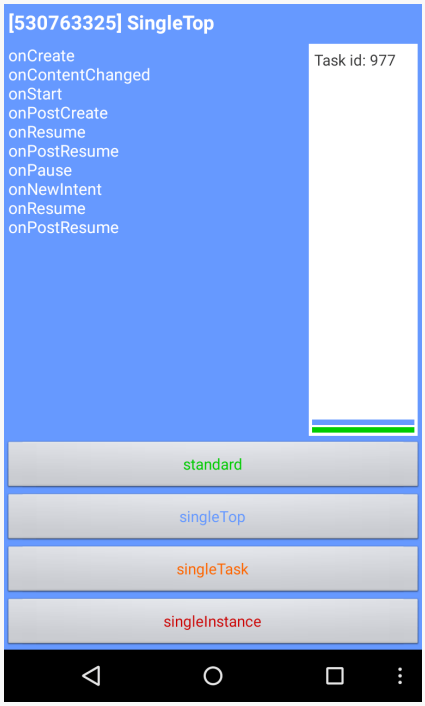


Figure 692: Novoda Demo App, After Starting singleTop Activity Again

Note that you can get a similar result by including `Intent.FLAG_ACTIVITY_SINGLE_TOP` on a `startActivity()` call. Using `launchMode` says you *always* want single-top behavior; using `FLAG_ACTIVITY_SINGLE_TOP` says that *this time* you want single-top behavior.

Pressing BACK returns you to the original green `standard` activity, with the blue `singleTop` activity having been destroyed.

singleTask

A `launchMode` of `singleTask` says that this activity must *always* be the root activity of a task.

If the task does not have that activity, a new task is created. So, if we tap the button to launch the `singleTask` activity in the Novoda demo app, we get a new task (ID 978, compared to the previous 977), with an instance of the yellow `singleTask` activity as its root:

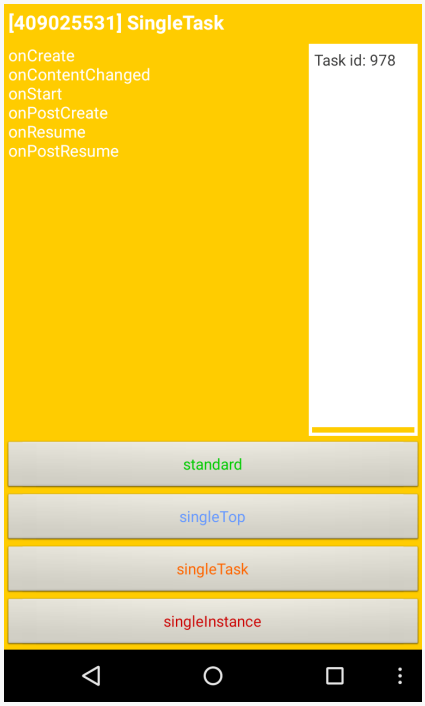


Figure 693: Novoda Demo App, After Starting singleTask Activity

However, if the activity in question is already there as the root of the task, all other activities on the back stack are cleared, and we are taken to the `singleTask` activity again.

So, in the Novoda demo app, if after we start the `singleTask` activity, we tap the button to launch a `standard` activity or two:

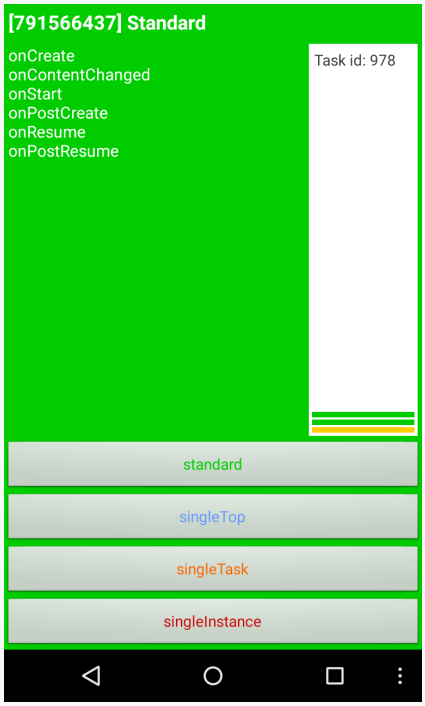


Figure 694: Novoda Demo App, Two standard Activities After singleTask Activity

...then tap the button to launch the `singleTask` activity, we get largely the same screen as before, just with a few more lifecycle methods logged:

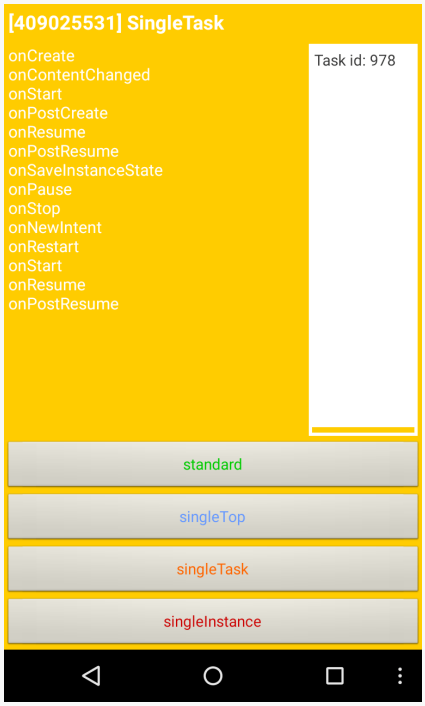


Figure 695: Novoda Demo App, After Starting singleTask Activity Again

It is the same task and the same instance, but with the other activities removed.

singleInstance

`singleInstance` works much like `singleTask`, except that the task will only ever hold this one activity. No other activities will be placed into the task.

So, tapping the button to start a `singleInstance` activity in the Novoda demo app brings up the red `singleInstance` UI:

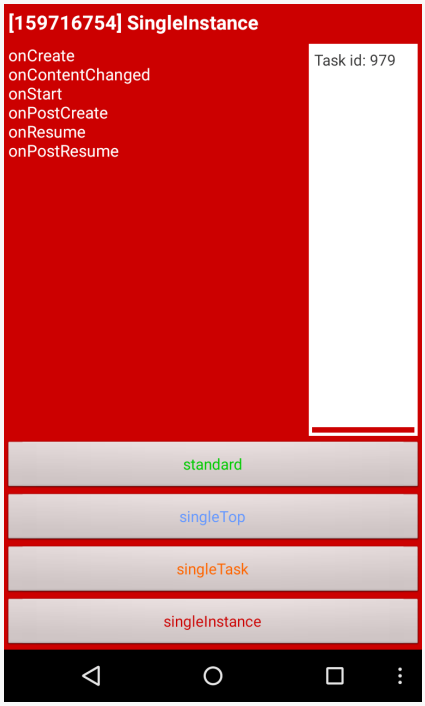


Figure 696: Novoda Demo App, After Starting singleInstance Activity

Tapping the same button again just triggers `onNewIntent()` and other lifecycle methods on the same activity in the same task. If, however, you try tapping on the button for the `standard` activity, your activity will go to another task. Depending on when and how you try the Novoda demo app, this could be a prior task associated with our app (e.g., one you used for earlier `standard` tests), or it could be a new task (if you do not have any other ones). This is based on the `taskAffinity` of the activity being started.

In general, `singleTask` and `singleInstance` are for unusual use cases, and ordinary Android apps should have little reason to use them. Google specifically urges you *not* to use them:

> ...standard is the default mode and is appropriate for most types of activities. SingleTop is also a common and useful launch mode for many types of activities. The other modes — singleTask and singleInstance — are not appropriate for most applications, since they result in an interaction model that is likely to be unfamiliar to users and is very different from most other applications.

alwaysRetainTaskState

As noted earlier in the chapter, tasks may be cleared by Android if the user has not been in the task for some time (e.g., 30+ minutes). In these cases, the user is taken back to the root activity.

If, however, the root activity has `android:alwaysRetainTaskState="true"` in its manifest entry, then Android will not apply this timeout rule. So long as the task exists, its entire state will be retained and used when the user returns to the task. This is useful for tasks where there is a lot of state that the user might regret losing.

Other Task-Related Activity Methods

There are a handful of other task-related methods and such floating around the `Activity` class:

finishAffinity()

This calls `finish()` not only on the current activity, but on all activities immediately behind it on the back stack for this task that have the same `taskAffinity` as does the current activity. Much of the time, the activities on the stack will all share an affinity, and therefore this will frequently finish all activities in the task. If the task has a mixed set of affinities (e.g., a mix of explicitly-named affinities and other activities using the default affinity), this method would only wipe out those behind the current with a specific match.

This method is not commonly used.

finishAndRemoveTask()

This calls `finish()` on all activities in the task and removes the task outright.

For example, a "logout" operation might call `finishAndRemoveTask()` to flush the current task, then call `startActivity()` to launch the login activity. That login activity will wind up in a fresh task (since the current one will be removed), and the old activity instances will go away, so the user cannot somehow stumble into them when they are not yet logged in.

getTaskId()

Returns a unique integer that identifies the task the activity resides in.

This method is not commonly used.

isTaskRoot()

`isTaskRoot()` is a method on `Activity`. It will return `true` if this activity instance is at the root of a task, meaning that pressing BACK should remove the task and return the user to the home screen.

moveTaskToBack()

This method moves the current task to the background. What comes to the foreground is undocumented but generally seems to be the task for the home screen. Some apps use this to offer a "minimize" or "go to background" option within the app, though this is superfluous, as the task will move to the background naturally as the user navigates their device.

setTaskDescription()

For Android 5.0+, `setTaskDescription()` allows you to associate an `ActivityManager.TaskDescription` instance with your task. Here you can provide values that help drive what the task looks like on the overview screen. Specifically, you can provide the icon, title, and background color to use for the title bar over your thumbnail on the overview screen.

The Assist API ("Now On Tap")

Android 6.0 introduced the concept of the device "assistant". The assistant can be triggered by a long-press of the HOME button or via a spoken phrase (if the user has always-on keyphrase detection) enabled. An assistant is a special app that has access to the content of the foreground activity and other visible windows, much like an accessibility service does.

For the vast majority of users of Google Play ecosystem devices running Android 6.0 or higher, the "assistant" is known as Now On Tap. On some devices, such as the Google Pixel series, this assistant is known simply as the "Google Assistant". This is marketed as an extension of the Google Now UI, where Now On Tap/Google Assistant will take the data from the foreground activity and use that to find other relevant things for the user to do based upon that data.

(for the purposes of this chapter, this Google-supplied assistant will be referred to as "Now On Tap", to distinguish Google's assistant from assistants that others might write using these APIs)

For example, suppose the user receives a text message, suggesting dinner at a particular restaurant. The restaurant is merely named — no URL — and so the text messaging client would just display the name of the restaurant as part of the message. If the user invokes Now On Tap, Google will take the contents of this message (and anything else on the screen), and presumably send it to Google's servers, sending back things like details about the restaurant (e.g., URL to Web site, Google's scanned reviews of the restaurant, link to Google Maps for driving directions). Google's search engine technology would scan the data from the app, recognize that the restaurant name appears to be something significant, and give Now On Tap details of what to offer the user.

As with many things from Google, Now On Tap is very compelling and very much a privacy problem. Now On Tap is automatically installed and enabled on Android 6.0 devices — users have to go through some work to disable it. Users and app developers have limited ability to control Now On Tap, in terms of what data it collects and what it does with that data. On the other hand, certain apps (for which there are no privacy considerations) might wish to provide *more* data to Now On Tap, beyond what is visible in widgets, to help provide more context for Now On Tap to help users.

In this chapter, we will explore the Assist API, in terms of:

- what data gets collected
- how apps can add to that data
- how apps can block sensitive information from the assistant
- how to write your own assistant, as a Now On Tap replacement

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

What Data Gets Disclosed

Quite a bit of data is made available to Now On Tap or other assistants through the Assist API alone, as will be explored in this section.

Assistants are welcome to use other APIs as well, subject to standard Android permissions and such. So, for example, an app might not show the device's location, and therefore an assistant could not get the location from the Assist API, but the assistant could use `LocationManager` or the Play Services location API to find out the device's location.

There is also a risk of pre-installed assistants using undocumented means of getting at data beyond what the normal Android SDK would allow.

All that being said, assistants will get a lot of information about the currently-visible UI, just from what the Assist API provides.

Screenshot

Assistants can get a screenshot of the current screen contents — minus the status bar — when the user activated the assistant (e.g., long-pressed HOME). Developers can block this for select activities or other windows. Hence, an assistant cannot *assume* that it will get a screenshot, though frequently it will.

Presumably, the "vision" here is to use computer vision and other image recognition techniques on the screenshot to find things of interest. For example, the user might bring up Now On Tap for some activity that is showing a photo of a monument. The activity might not be showing any other details about the monument, such as its name. However, Google's servers might well recognize what monument it is and therefore give the user links to Wikipedia pages about the monument, a map of where the monument is located, etc.

View Structure

By far the largest dump of data that the assistant gets comes in the form of the view structure. This is represented by a tree of `AssistStructure.ViewNode` objects, one per widget or container within a window. These provide similar information as to what one gets from the accessibility APIs. For most assistants, the key data is the text or content description in the widget. In the case of text, this is available as a `CharSequence` and so may contain additional information (e.g., hyperlinks represented in `URLSpan` objects) beyond the words visible to the user.

Developers can restrict what widgets and containers are disclosed, but that is something developers have to do explicitly. In other words, making data available to assistants is something a developer has to opt out of, not opt into.

Other Data

In addition to the view structure and a largely-undocumented `Bundle`, the other piece of data supplied to the assistant is the `AssistContent`. Here is where an app can provide some additional context about the foreground activity.

Specifically, the app can provide:

- an `Intent` that represents the activity, replacing the `Intent` that was used to start the activity, if there is a better one for long-term use (e.g., the activity

was started via a `Notification` action and you want to route the user through a different `Intent` for other scenarios)
- a `Uri` that points to some Web page of relevance for this activity
- a string of "structured data", designed to be populated by a snippet of JSON using the schema.org specification, to provide details of the book, song, video, or whatever happens to be in the activity at the moment
- another undocumented `Bundle`
- an undocumented `ClipData`

Assistants can use this directly (e.g., offer a link to the `Uri` supplied in this content) or indirectly (e.g., using the schema.org JSON to find places where the user can purchase related content).

Adding to the Data

You may wish to provide some additional information to Now On Tap or other assistants, such as the `Intent` or JSON described above. Or, you may just generally want to ensure that your app provides the maximum amount of information to these assistants, without necessarily trying to invent new data to provide.

There are a few options for accomplishing this.

Accessibility

The big one is to ensure that your app provides text or content descriptions for everything visible. This will not only help these assistants, but this will make your app far more accessible to those using TalkBack or other accessibility services.

Mostly, this is a matter of ensuring that your `ImageView` widgets and other non-textual widgets have a content description, whether set via `android:contentDescription` attributes or by `setContentDescription()` in Java. `TextView` and its subclasses automatically use their text as the content description; `EditText` will use the hint if there is no text in the field at the moment.

More advice regarding accessibility can be found in the chapter on accessibility and focus management.

Assist-Specific Data

Beyond that, you can contribute to the `AssistContent` (where the `Intent`, `Uri`, and JSON live) and other assist-related information for a given invocation of the assistant by the user.

You have a few options of where to place this logic: in one spot globally, on a per-activity basis, and, for custom views, on a per-view basis.

Globally

You can call `registerOnProvideAssistDataListener()` on the global `Application` object (retrieved by calling `getApplicationContext()` on some other `Context`, like your `Activity`). This takes an `OnProvideAssistDataListener` implementation, which in turn provides an `onProvideAssistData()` implementation, that will be called when the assistant is requested. You are passed the `Activity` of yours that is in the foreground, along with a `Bundle` that you can fill in.

However, the documentation only says that the `Bundle` will go into the `EXTRA_ASSIST_CONTEXT` extra on the `Intent` that invokes the assistant. What that `Bundle` is supposed to contain is undocumented.

Per-Activity

Your primary hooks for customizing the assist data come in the form of two callbacks on your `Activity` subclasses: `onProvideAssistData()` and `onProvideAssistContent()`.

`onProvideAssistData()` is given the same `Bundle` that is given to the `OnProvideAssistDataListener` on a global basis. However, it is unclear what goes in that `Bundle`, and the contents of that `Bundle` do not appear to make it to the assistant, at least through the documented Assist API.

`onProvideAssistContent()`, though, is more relevant.

The `Assist/MoAssist` sample project is a clone of a sample app demonstrating the use of tabs in Android. The clone has its `compileSdkVersion` bumped to 23, and it overrides `onProvideAssistData()` and `onProvideAssistContent()`:

```
@Override
public void onProvideAssistData(Bundle data) {
```

```
    super.onProvideAssistData(data);

    data.putInt("random-value", new SecureRandom().nextInt());
  }

  @TargetApi(23)
  @Override
  public void onProvideAssistContent(AssistContent outContent) {
    super.onProvideAssistContent(outContent);

    outContent.setWebUri(Uri.parse("https://commonsware.com"));

    try {
      JSONObject json=new JSONObject()
        .put("@type", "Book")
        .put("author", "https://commonsware.com/mmurphy")
        .put("publisher", "CommonsWare, LLC")
        .put("name", "The Busy Coder's Guide to Android Development");

      outContent.setStructuredData(json.toString());
    }
    catch (JSONException e) {
      Log.e(getClass().getSimpleName(),
        "Um, what happened here?", e);
    }
  }
```

(from Assist/MoAssist/app/src/main/java/com/commonsware/android/assist/mo/MainActivity.java)

The `onProvideAssistData()` simply puts a random number into the `Bundle`. That random number does not appear anywhere in the data collected by an assistant.

`onProvideAssistContent()` fills in two items in the `AssistContent`:

- a Web URL of relevance to the activity, in this case the home page of the book's publisher
- a bit of JSON, following the published schema.org `Book` structure, with metadata about this book

This information *is* supplied to assistants and can be used by them to do something useful, such as offer links for the user to click on to visit the sites.

Per-View

If you are implementing your own custom views, particularly those that render their own text using low-level `Canvas` APIs, you may wish to override `onProvideStructure()` and/or `onProvideVirtualStructure()`. These will be called on your widgets to provide the `AssistStructure.ViewNode` details to be passed to the assistant.

However, in all likelihood, you would want to instead work with the accessibility APIs to publish data to be used by accessibility services, such as the text that you are rendering. If you do that, the default implementations of `onProvideStructure()` and `onProvideVirtualStructure()` should suffice.

Removing from the Data

While some developers may embrace Now On Tap, others may specifically want to prevent Now On Tap or other assistants from "spying" on application data. You have a few options for controlling what is provided to assistants; however, all require work and some have side effects. For example, there is nothing in the manifest that you can specify to make your activities opt out of providing assist data.

FLAG_SECURE

The standard approach for making private activities *really* private is to use `FLAG_SECURE`:

```
public class FlagSecureTestActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    getWindow().setFlags(LayoutParams.FLAG_SECURE,
                         LayoutParams.FLAG_SECURE);

    setContentView(R.layout.main);
  }
}
```

Call `setFlags()` before `setContentView()`, in this case setting `FLAG_SECURE`.

The classic effect of FLAG_SECURE is to block screenshots, both user-initiated ones and system-initiated ones (e.g., the screenshots used in the overview/recent-tasks screen on Android 4.0+).

If the user triggers an assistant for a secure activity, the assistant will not get the full view structure (i.e., no widgets and no text) and will not get a screenshot.

Password Fields

An EditText that is set up as a password field will have its text blocked from the view structure. The widget will be listed, but its text will be null.

Presumably, this relies on the EditText using a PasswordTransformationMethod, as that is Android's typical approach for determining whether or not an EditText is deemed to be secure. If you have implemented your own TransformationMethod (e.g., with a different approach for shrouding the user input), either have it extend PasswordTransformationMethod or use other approaches to prevent this field's contents from being published to assistants.

NoAssistFrameLayout

The apparently-official way to block a widget or container from participating in the assist API is to create a subclass of it and override dispatchProvideStructure(). The stock implementation of this triggers the calls to onProvideStructure() and onProvideVirtualStructure(). Plus, for a ViewGroup, it will iterate over the children and call dispatchProvideStructure() on each of them.

If you are creating your own custom view, and you want it eliminated from the view structure, just override dispatchProvideStructure() and have it do nothing.

Or, you can create a container that is there solely to block the assist data collection. The Assist/NoAssist sample project does this, in the form of a NoAssistFrameLayout:

```
package com.commonsware.android.assist.no;

import android.annotation.TargetApi;
import android.content.Context;
import android.os.Build;
import android.util.AttributeSet;
import android.view.ViewStructure;
import android.widget.FrameLayout;
```

```
public class NoAssistFrameLayout extends FrameLayout {
  public NoAssistFrameLayout(Context context) {
    super(context);
  }

  public NoAssistFrameLayout(Context context,
                             AttributeSet attrs) {
    super(context, attrs);
  }

  public NoAssistFrameLayout(Context context,
                             AttributeSet attrs,
                             int defStyleAttr) {
    super(context, attrs, defStyleAttr);
  }

  @TargetApi(Build.VERSION_CODES.LOLLIPOP)
  public NoAssistFrameLayout(Context context,
                             AttributeSet attrs,
                             int defStyleAttr,
                             int defStyleRes) {
    super(context, attrs, defStyleAttr, defStyleRes);
  }

  @Override
  public void dispatchProvideStructure(ViewStructure structure) {
    // no, thanks
  }
}
```

(from Assist/NoAssist/app/src/main/java/com/commonsware/android/assist/no/NoAssistFrameLayout.java)

`EditorFragment` — responsible for showing a large multi-line `EditText` for the user to type into — will conditionally use a `NoAssistFrameLayout`, specifically on the third tab (a `ViewPager` `position` of 2):

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    int position=getArguments().getInt(KEY_POSITION, -1);
    View result;

    if (position==2) {
      ViewGroup doctorNo=new NoAssistFrameLayout(getActivity());
      inflater.inflate(R.layout.editor, doctorNo);
```

```
      result=doctorNo;
    }
    else {
      result=inflater.inflate(R.layout.editor, container, false);
    }

    EditText editor=(EditText)result.findViewById(R.id.editor);

    editor.setHint(getTitle(getActivity(), position));

    if (position==1) {
      editor.
        setTransformationMethod(PasswordTransformationMethod.
          getInstance());
    }

    return(result);
  }
```

(from Assist/NoAssist/app/src/main/java/com/commonsware/android/assist/no/EditorFragment.java)

If we are on the third tab, we create a `NoAssistFrameLayout` and inflate our `EditText` into it. Otherwise, we inflate the layout normally.

Note that this sample also applies a `PasswordTransformationMethod` for the second page of the `ViewPager` (a `position` of 1), to illustrate the `null` text that will be recorded as a result.

Blocking Assist as a User

It is possible that your reaction to all of this is that you want to opt out of Now On Tap as a user. Or, perhaps you want to provide some instructions to your users on how to opt out of Now On Tap.

Go to Settings > Apps. There should be an option for advanced app configuration actions (on Nexus-series devices, this is a gear icon in the action bar). Tap that, then choose "Default Apps" to bring up categories of default apps for various actions:

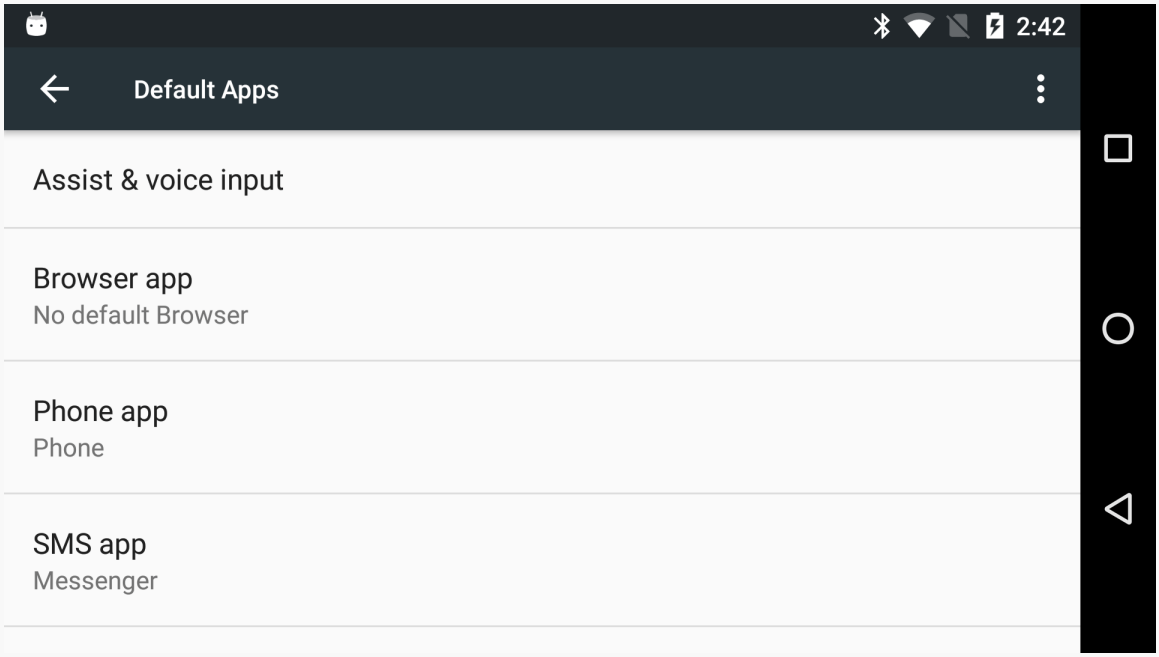


Figure 697: Android 6.0 Default Apps Screen in Settings

In there, tap on "Assist & voice input". By default, you should see "Google App" as the chosen option, which means that Now On Tap is active:

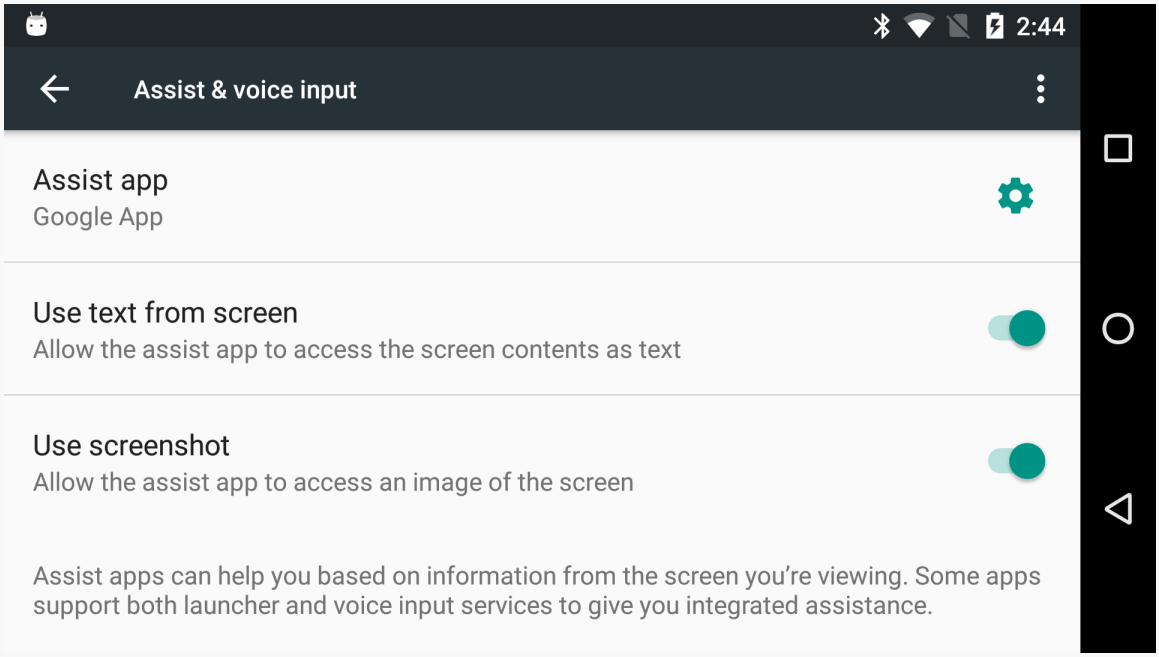


Figure 698: Android 6.0 Assist & Voice Input Screen in Settings

Tapping on that entry will bring up a list of available options, including "None":

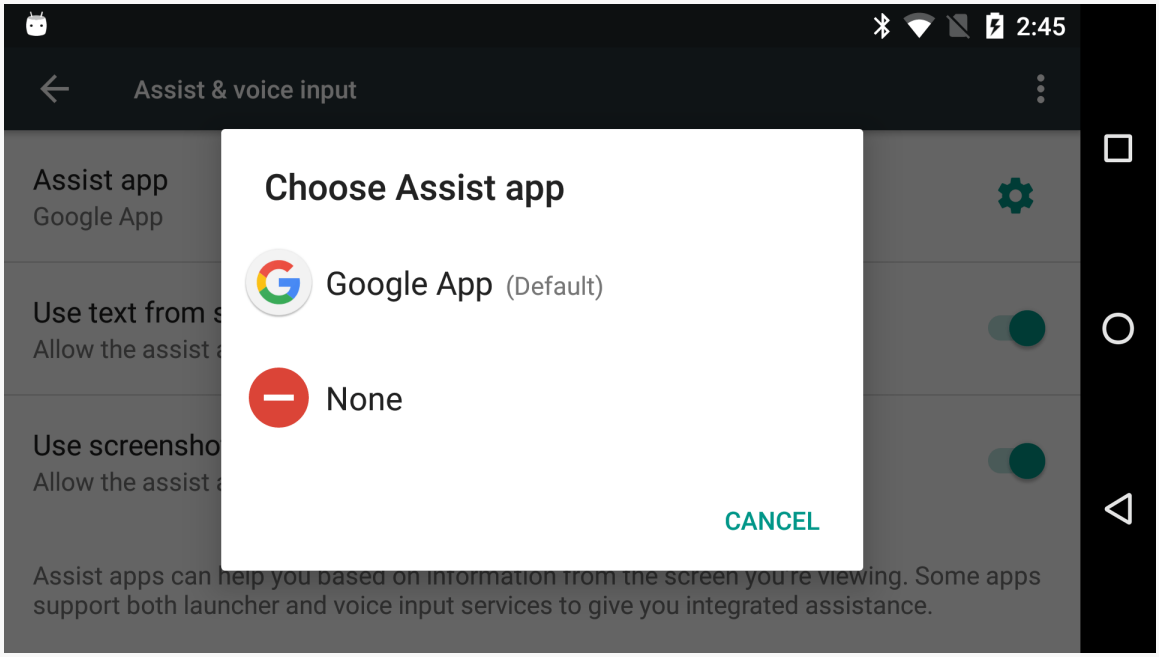


Figure 699: Android 6.0 Assist & Voice Input Options in Settings

Implementing Your Own Assistant

While Now On Tap is pre-installed and pre-activated, and while users can disable Now On Tap, another option for users is to activate some other assistant. Any app that implements the proper pieces of the Assist API will appear in the roster of available assistants for the user to choose from, as described in the previous section. The `Assist/AssistLogger` sample project represents one such app.

Primarily, this app is for diagnostic purposes, showing you exactly what your activity is "leaking" to assistants. It was essential in figuring out how the APIs shown in earlier examples in this chapter worked, for instance. However, it also serves as a demonstration of the minimum requirements to implement an assistant in general.

Creating an assistant is technically part of a larger bit of work on handling voice interactions in Android. However, if all you want is an assistant, you can ignore the voice-related bits.

A Stub VoiceInteractionService

Some of what is needed to set up an assistant is some boilerplate.

For example, the entry point for assistants and voice interactions is a custom subclass of `VoiceInteractionService`. If you only are concerned with implementing an assistant, your `VoiceInteractionService` can be empty:

```
package com.commonsware.android.assist.logger;

import android.service.voice.VoiceInteractionService;

public class AssistLoggerService extends VoiceInteractionService {
}
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistLoggerService.java)

However, it needs to exist, and in particular it needs to have its `<service>` entry in your manifest:

```
    <service
      android:name=".AssistLoggerService"
      android:permission="android.permission.BIND_VOICE_INTERACTION">
      <meta-data
        android:name="android.voice_interaction"
        android:resource="@xml/assist_service"/>
```

```
      <intent-filter>
        <action android:name="android.service.voice.VoiceInteractionService"/>
      </intent-filter>
    </service>
```

(from Assist/AssistLogger/app/src/main/AndroidManifest.xml)

The keys to the manifest entry are:

- It needs to have the `android:permission` attribute, limiting it clients that hold the `BIND_VOICE_INTERACTION` permission, which should limit clients to those that are part of the device firmware
- It needs to have the `<intent-filter>` advertising that it supports the `android.service.voice.VoiceInteractionService` action string
- It needs an `android.voice_interaction` `<meta-data>` element, pointing to an XML resource that further configures the voice interaction/assistant implementation

The sample project has that metadata in `res/xml/assist_service.xml`:

```
<voice-interaction-service
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:recognitionService="com.commonsware.android.assist.logger.AssistLoggerService"
  android:sessionService="com.commonsware.android.assist.logger.AssistLoggerSessionService"
  android:supportsAssist="true"/>

<!--
  android:settingsActivity="com.android.test.voiceinteraction.SettingsActivity"
-->
```

(from Assist/AssistLogger/app/src/main/res/xml/assist_service.xml)

There are three attributes required on the `<voice-interaction-service>` root element to enable an assistant:

- `android:recognitionService` points back to your `VoiceInteractionService` subclass
- `android:sessionService` points to a subclass of `VoiceInteractionSessionService` (we will examine the project's implementation shortly)
- `android:supportsAssist` should be `true`

If you want, you can also have an `android:settingsActivity` attribute, shown in this XML as a commented-out snippet at the end of the file. This can point to an activity in your app. If you have this, a gear icon will appear on the "Assist & voice

input" Settings screen that, when tapped, will bring up this activity, to configure the behavior of your assistant. The sample app skips this.

A Trivial VoiceInteractionSessionService

The service pointed to by `android:sessionService` in the metadata needs to be a subclass of `VoiceInteractionSessionService`. The only method that you need to override is `onNewSession()`, where you can return an instance of a `VoiceInteractionSession`:

```
package com.commonsware.android.assist.logger;

import android.os.Bundle;
import android.service.voice.VoiceInteractionSession;
import android.service.voice.VoiceInteractionSessionService;

public class AssistLoggerSessionService extends
  VoiceInteractionSessionService {
  @Override
  public VoiceInteractionSession onNewSession(Bundle args) {
    return(new AssistLoggerSession(this));
  }
}
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistLoggerSessionService.java)

Here, we return an instance of `AssistLoggerSession`, which is where all of our real business logic resides for our assistant.

Note that this service also should use `android:permission` to limit clients to those that hold the `android.permission.BIND_VOICE_INTERACTION` permission:

```
    <service
      android:name=".AssistLoggerSessionService"
      android:permission="android.permission.BIND_VOICE_INTERACTION"/>
```

(from Assist/AssistLogger/app/src/main/AndroidManifest.xml)

The VoiceInteractionSession

`VoiceInteractionSession` has a *lot* of methods that you can override, both for voice interactions and for assistant invocations. The sample app overrides the minimum required for an assistant, as its mission simply is to log all of the data received by our assistant to files on external storage, for diagnostic purposes.

NOTE: Running this sample app on hardware that is actually used with private data is stupid beyond words. Any app can then read the files on external storage and see what information is published by whatever apps are in the foreground at the times when you invoke the assistant. Please use this only on test environments.

Basic Setup

Akin to components, a `VoiceInteractionSession` has an `onCreate()` method, called as part of setting up the session. In there, `AssistLoggerSession` sets up an output directory for logging the results, assuming that external storage is available:

```
@Override
public void onCreate() {
  super.onCreate();

  if (Environment.MEDIA_MOUNTED
    .equals(Environment.getExternalStorageState())) {
    String logDirName=
      "assistlogger_"+
       new SimpleDateFormat("yyyyMMdd'-'HHmmss").format(new Date());

    logDir=
      new File(getContext().getExternalCacheDir(), logDirName);
    logDir.mkdirs();
  }
}
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistLoggerSession.java)

onHandleScreenshot()

If the user invokes your assistant, you will be called with `onHandleScreenshot()`. Usually, you will be passed a `Bitmap` that contains the screenshot. However, if the foreground activity is using `FLAG_SECURE`, the `Bitmap` that is passed to you will be `null`, so make sure you check it before doing anything with it.

The `AssistLoggerSession` forks a `ScreenshotThread` to save this screenshot in the background:

```
@Override
public void onHandleScreenshot(Bitmap screenshot) {
  super.onHandleScreenshot(screenshot);
```

```
    if (screenshot!=null) {
      new ScreenshotThread(getContext(), logDir, screenshot).start();
    }
  }
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistLoggerSession.java)

`ScreenshotThread`, in turn, just uses `compress()` on `Bitmap` to write the image out as a PNG to the directory that we are using for logging:

```
  private static class ScreenshotThread extends Thread {
    private final File logDir;
    private final Bitmap screenshot;
    private final Context ctxt;

    ScreenshotThread(Context ctxt, File logDir, Bitmap screenshot) {
      this.ctxt=ctxt.getApplicationContext();
      this.logDir=logDir;
      this.screenshot=screenshot;
    }

    @Override
    public void run() {
      if (logDir!=null) {
        try {
          File f=new File(logDir, "screenshot.png");
          FileOutputStream fos=new FileOutputStream(f);

          screenshot.compress(Bitmap.CompressFormat.PNG, 100, fos);
          fos.flush();
          fos.getFD().sync();
          fos.close();

          MediaScannerConnection
            .scanFile(ctxt,
              new String[] {f.getAbsolutePath()},
              new String[] {"image/png"}, null);

          Log.d(getClass().getSimpleName(),
            "screenshot written to: "+f.getAbsolutePath());
        }
        catch (IOException e) {
          Log.e(getClass().getSimpleName(),
            "Exception writing out screenshot", e);
        }
      }
      else {
        Log.d(getClass().getSimpleName(),
```

```
          String.format("onHandleScreenshot: %dx%d",
            screenshot.getWidth(), screenshot.getHeight()));
      }
    }
  }
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistLoggerSession.java)

onHandleAssist()

`onHandleAssist()` is your other main assistant callback. Here is where you get:

- a `Bundle` of undocumented stuff
- the `AssistStructure` outlining the contents of the windows, including the view hierarchy
- the `AssistContent` with the `Intent`, Web `Uri`, JSON, and so on

`AssistLoggerSession` kicks off an `AssistDumpThread` to record this data in the background:

```
  @Override
  public void onHandleAssist(Bundle data,
                             AssistStructure structure,
                             AssistContent content) {
    super.onHandleAssist(data, structure, content);

    new AssistDumpThread(getContext(), logDir, data, structure,
      content).start();
  }
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistLoggerSession.java)

`AssistDumpThread` itself is a *long* class that generates a JSON file containing the information found in the parameters to `onHandleAssist()`:

```
package com.commonsware.android.assist.logger;

import android.app.assist.AssistContent;
import android.app.assist.AssistStructure;
import android.content.Context;
import android.content.Intent;
import android.media.MediaScannerConnection;
import android.os.Bundle;
import android.util.Log;
import org.json.JSONArray;
import org.json.JSONException;
```

```
import org.json.JSONObject;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Set;

class AssistDumpThread extends Thread {
  private final File logDir;
  private final Bundle data;
  private final AssistStructure structure;
  private final AssistContent content;
  private final Context ctxt;

  AssistDumpThread(Context ctxt, File logDir, Bundle data,
                   AssistStructure structure,
                   AssistContent content) {
    this.ctxt=ctxt.getApplicationContext();
    this.logDir=logDir;
    this.data=data;
    this.structure=structure;
    this.content=content;
  }

  @Override
  public void run() {
    if (logDir!=null) {
      JSONObject json=new JSONObject();

      try {
        json.put("data", dumpBundle(data, new JSONObject()));
      }
      catch (JSONException e) {
        Log.e(getClass().getSimpleName(),
          "Exception saving data", e);
      }

      try {
        json.put("content", dumpContent(new JSONObject()));
      }
      catch (JSONException e) {
        Log.e(getClass().getSimpleName(),
          "Exception saving content", e);
      }

      try {
        json.put("structure", dumpStructure(new JSONObject()));
      }
```

```
    catch (JSONException e) {
      Log.e(getClass().getSimpleName(),
        "Exception saving structure", e);
    }

    File f=new File(logDir, "assist.json");

    try {
      FileOutputStream fos=new FileOutputStream(f);
      OutputStreamWriter osw=new OutputStreamWriter(fos);
      PrintWriter pw=new PrintWriter(osw);

      pw.print(json.toString(2));
      pw.flush();
      fos.getFD().sync();
      fos.close();

      MediaScannerConnection
        .scanFile(ctxt,
          new String[] {f.getAbsolutePath()},
          new String[] {"application/json"}, null);

      Log.d(getClass().getSimpleName(),
        "assist data written to: "+f.getAbsolutePath());
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
        "Exception writing out assist data", e);
    }
  }
  else {
    Log.d(getClass().getSimpleName(), "onHandleAssist");
  }
}

JSONObject dumpBundle(Bundle b, JSONObject json)
  throws JSONException {
  Set<String> keys=b.keySet();

  for (String key : keys) {
    json.put(key, wrap(b.get(key)));
  }

  return (json);
}

private JSONObject dumpContent(JSONObject json)
  throws JSONException {
```

```
    JSONObject extras=new JSONObject();

    if (content.getExtras()!=null) {
      json.put("extras", extras);
      dumpBundle(content.getExtras(), extras);
    }

    if (content.getIntent()!=null) {
      json.put("intent",
        content.getIntent().toUri(Intent.URI_INTENT_SCHEME));
    }

    json.put("structuredData",
      wrap(content.getStructuredData()));
    json.put("webUri", wrap(content.getWebUri()));

    return (json);
  }

  private JSONObject dumpStructure(JSONObject json)
    throws JSONException {
    return (json.put("windows",
      dumpStructureWindows(new JSONArray())));
  }

  private JSONArray dumpStructureWindows(JSONArray windows)
    throws JSONException {
    for (int i=0; i<structure.getWindowNodeCount(); i++) {
      windows.put(
        dumpStructureWindow(structure.getWindowNodeAt(i),
          new JSONObject()));
    }

    return (windows);
  }

  private JSONObject dumpStructureWindow(
    AssistStructure.WindowNode window,
    JSONObject json)
    throws JSONException {
    json.put("displayId", wrap(window.getDisplayId()));
    json.put("height", wrap(window.getHeight()));
    json.put("left", wrap(window.getLeft()));
    json.put("title", wrap(window.getTitle()));
    json.put("top", wrap(window.getTop()));
    json.put("width", wrap(window.getWidth()));
    json.put("root",
      dumpStructureNode(window.getRootViewNode(),
```

```
        new JSONObject()));

    return (json);
  }

  private JSONObject dumpStructureNode(
    AssistStructure.ViewNode node,
    JSONObject json)
    throws JSONException {
    json.put("accessibilityFocused",
      wrap(node.isAccessibilityFocused()));
    json.put("activated", wrap(node.isActivated()));
    json.put("alpha", wrap(node.getAlpha()));
    json.put("assistBlocked", wrap(node.isAssistBlocked()));
    json.put("checkable", wrap(node.isCheckable()));
    json.put("checked", wrap(node.isChecked()));
    json.put("className", wrap(node.getClassName()));
    json.put("clickable", wrap(node.isClickable()));
    json.put("contentDescription",
      wrap(node.getContentDescription()));
    json.put("contextClickable",
      wrap(node.isContextClickable()));
    json.put("elevation", wrap(node.getElevation()));
    json.put("enabled", wrap(node.isEnabled()));

    if (node.getExtras()!=null) {
      json.put("extras", dumpBundle(node.getExtras(),
        new JSONObject()));
    }

    json.put("focusable", wrap(node.isFocusable()));
    json.put("focused", wrap(node.isFocused()));
    json.put("height", wrap(node.getHeight()));
    json.put("hint", wrap(node.getHint()));
    json.put("id", wrap(node.getId()));
    json.put("idEntry", wrap(node.getIdEntry()));
    json.put("idPackage", wrap(node.getIdPackage()));
    json.put("idType", wrap(node.getIdType()));
    json.put("left", wrap(node.getLeft()));
    json.put("longClickable", wrap(node.isLongClickable()));
    json.put("scrollX", wrap(node.getScrollX()));
    json.put("scrollY", wrap(node.getScrollY()));
    json.put("isSelected", wrap(node.isSelected()));
    json.put("text", wrap(node.getText()));
    json.put("textBackgroundColor",
      wrap(node.getTextBackgroundColor()));
    json.put("textColor", wrap(node.getTextColor()));
    json.put("textLineBaselines",
```

```
      wrap(node.getTextLineBaselines()));
    json.put("textLineCharOffsets",
      wrap(node.getTextLineCharOffsets()));
    json.put("textSelectionEnd",
      wrap(node.getTextSelectionEnd()));
    json.put("textSelectionStart",
      wrap(node.getTextSelectionStart()));
    json.put("textSize", wrap(node.getTextSize()));
    json.put("textStyle", wrap(node.getTextStyle()));
    json.put("top", wrap(node.getTop()));
    json.put("transformation",
      wrap(node.getTransformation()));
    json.put("visibility", wrap(node.getVisibility()));
    json.put("width", wrap(node.getWidth()));

    json.put("children",
      dumpStructureNodes(node, new JSONArray()));

    return (json);
  }

  private JSONArray dumpStructureNodes(
    AssistStructure.ViewNode node,
    JSONArray children) throws JSONException {
    for (int i=0; i<node.getChildCount(); i++) {
      children.put(dumpStructureNode(node.getChildAt(i),
        new JSONObject()));
    }

    return (children);
  }

  private Object wrap(Object thingy) {
    if (thingy instanceof CharSequence) {
      return (JSONObject.wrap(thingy.toString()));
    }

    return (JSONObject.wrap(thingy));
  }
}
```

(from Assist/AssistLogger/app/src/main/java/com/commonsware/android/assist/logger/AssistDumpThread.java)

Making a Real Assistant

`AssistLogger` is a faithful implementation of an assistant, but it does not really assist the user, except in seeing what sorts of information Google gets via Now On Tap.

If you wanted to make an actual assistant that is a true replacement for Now On Tap, you would also need to implement methods like:

- `onCreateContentView()`, where you can inflate a layout or otherwise assemble the basic UI to be shown to the user when your assistant is invoked
- `onShow()`, where you can populate that UI with the details for this particular assist request
- `onHide()`, called when your UI is no longer visible to the user

...and so on.

Determining the Active Assistant

If you elect to create your own assistant, you might be interested in knowing whether or not your app has been chosen as the user's assistant. Unfortunately, there is no documented and supported means of doing this.

So, here is the undocumented and unsupported approach that works on Android 6.0.

WARNING: this code may not work on all Android 6.0 devices, let alone on future versions of Android, as it relies a bit on internal implementation that could be changed by device manufacturers or custom ROM authors. Please use this *very carefully* and do not be shocked if it stops working.

`Settings.Secure` holds the details of the currently-chosen assistant. However, the key under which those details are stored is a hidden entry in `Settings.Secure`, and so it does not show up in the Android SDK. The key is `"voice_interaction_service"`. The value is the `ComponentName` of the assistant, serialized (or "flattened") into a `String`. So, to get the `ComponentName` of the assistant, you can use:

```
String assistant=
  Settings.Secure.getString(getContentResolver(),
    "voice_interaction_service");
```

```
boolean areWeGood=false;

if (assistant!=null) {
  ComponentName cn=ComponentName.unflattenFromString(assistant);
}
```

`cn` will then hold the `ComponentName`.

Leading the User to Make an Assistant Change

If you implement your own assistant, and at the moment you are not the user's chosen assistant, you might have the need to lead the user over to the spot in the Settings app where they can change this. Once again, this is not explicitly documented.

However, for Android 6.0, `Settings.ACTION_VOICE_INPUT_SETTINGS` contains the action string that opens up the screen where the user can choose their assistant implementation. So, you could call:

```
startActivity(new Intent(Settings.ACTION_VOICE_INPUT_SETTINGS));
```

to lead the user to that screen, plus use a `Toast` or something to remind the user to tap on the "Assist app" entry to choose the assistant.

However:

- Since `Settings.ACTION_VOICE_INPUT_SETTINGS` is not guaranteed to be on all devices, please wrap the `startActivity()` call in an `ActivityNotFoundException` `try/catch` block and deal with the missing action accordingly
- There is no guarantee that `Settings.ACTION_VOICE_INPUT_SETTINGS` will lead the user to the correct screen on all Android 6.0+ devices, as the Settings app might be altered by the device manufacturer or custom ROM author

The Autofill API

Many Web browsers, such as Chrome, offer "autofill". The browser remembers things that you have typed into certain fields, like addresses, and offers to fill those back in when you go to fill in the same form again... or even a different form with similar fields.

Android 8.0 adds autofill capability to Android via the Autofill Framework. Values that users enter into your forms will be remembered and offered again when the user returns to your form, or to other forms that they encounter in other apps on the device. This all happens without any changes to your code... in theory. In practice, you may need some changes to your code.

However, Android itself does not store the form data, nor offer it as suggestions for autofilling in a form. Android is merely serving as a conduit between your app and a third-party autofill service. It is the service that is responsible for determining what could be autofilled in your form and saving prior entries. As with the Assist API, the user can choose an autofill service, and none ship by default with the O Developer Preview. In all likelihood, a Google-branded autofill implementation will be part of the production rollout of Android 8.0, but the user could choose other autofill apps... or perhaps none at all, for privacy reasons.

In this chapter, we will explore the AutoFill API, in terms of:

- what data gets collected
- how apps can block sensitive information from autofill
- how apps can better integrate with autofill

NOTE: O Developer Preview 2 renamed many of the Autofill Framework classes and methods to make the `f` in `Autofill` lowercase. So, for example, `AutoFillService`

became `AutofillService`. Materials written based on O Developer Preview 1 (blog posts, Stack Overflow answers, etc.) will have a capital `F` instead.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

There are commonalities between autofill and the Assist API, and so this chapter will refer back to that one from time to time.

The Pieces of the Puzzle

Android 8.0 provides a framework for apps with a UI to participate in autofill, by denoting roles that some of those widgets have (e.g., "this is the field for the username"). However, Android itself does not save any user input for later use with autofill, nor does Android itself decide what values are candidates for autofill. Instead, Android delegates that work to a separate autofill service. This works akin to how accessibility services and input method editors work:

- Android provides the conduit for apps and pluggable service implementations to communicate
- Android provides a UI in Settings for the user to choose which service implementation(s) to use for a particular role
- App developers provide the actual service implementations

Hence, with autofill, there are three pieces: the app with the UI, Android itself, and the autofill service:

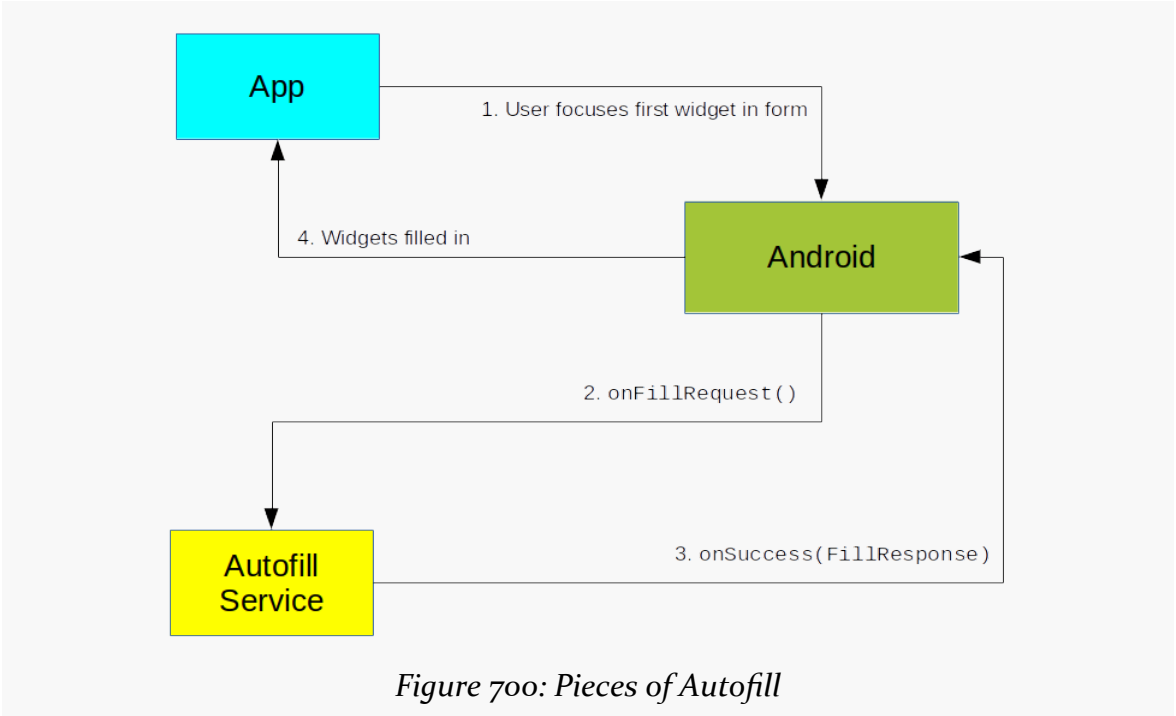


Figure 700: Pieces of Autofill

Here, `onFillRequest()` and `onSuccess()` are part of the API for autofill service implementations.

The User Experience

Before we get into the implementation, let's see how this looks from the standpoint of the user.

NOTE: These screenshots and explanations are from the O Developer Preview 4. It is possible that this flow will change by the time Android O ships in final form, or even in future developer previews.

While Completing a Form

If an autofill service is activated, and the user visits a screen with a form to be filled in, the user may get a drop-down list from the form, indicating possible "datasets" to use to complete the form:

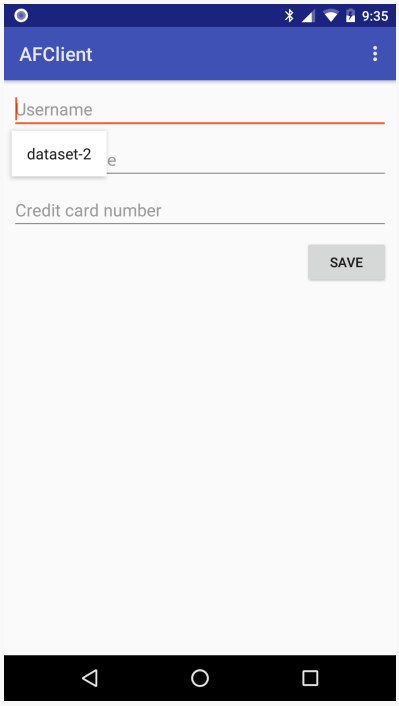


Figure 701: Autofill Datasets for User to Choose From

The actual presentation here ("dataset-2") comes from the autofill service. It is actually a `RemoteViews`, not just some text, and so different autofill services will use different formats here.

After Completing a Form

After the user has completed a form, with data not represented in one of those datasets, once the activity containing the form is destroyed, the user gets a "bottom sheet" panel asking if the data should be saved for later autofill use:

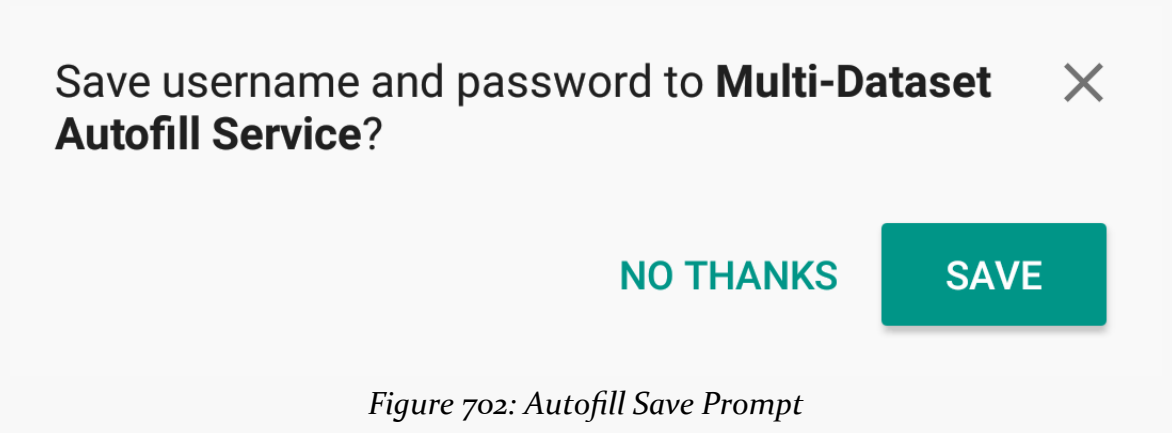


Figure 702: Autofill Save Prompt

This works akin to the pop-up dialog a desktop Web browser might use to ask if certain data should be saved (e.g., authentication credentials) for future visits to that page.

What Data Gets Disclosed

The Assist API discloses quite a bit of data to the user's chosen assistant app.

The autofill API discloses a bit less. Mostly, it is the view hierarchy, in the form of a `ViewStructure`.

An autofill service gets information about what is in the UI at two points:

1. When it is time to determine if there is anything that could be autofilled-in
2. When it is time to save data for future autofill requests

Here is what an autofill service might see from a small form with `username`, `password`, and `cc` fields (the latter for a credit card number), along with related labels and a "save" button, when it is time to determine if there is anything that could be autofilled-in:

```
{
  "data": {
    "logDirName":
"\/storage\/emulated\/0\/Android\/data\/com.commonsware.android.autofill.logger\/cache\/autofilllogger_
  },
  "structures": [
    {
      "windows": [
```

```
       {
         "displayId": 0,
         "height": 1920,
         "left": 0,
         "title":
"com.commonsware.android.autofill.client\/com.commonsware.android.autofill.client.MainActivity",
         "top": 0,
         "width": 1080,
         "root": {
           "accessibilityFocused": false,
           "activated": false,
           "alpha": 1,
           "assistBlocked": false,
           "autofillHints": null,
           "autofillOptions": null,
           "autofillType": 0,
           "autofillValue": null,
           "checkable": false,
           "checked": false,
           "className": "android.widget.FrameLayout",
           "clickable": false,
           "contentDescription": null,
           "contextClickable": false,
           "elevation": 0,
           "enabled": true,
           "focusable": false,
           "focused": false,
           "height": 1920,
           "hint": null,
           "htmlInfo": null,
           "id": -1,
           "idEntry": null,
           "idPackage": null,
           "idType": null,
           "inputType": 0,
           "left": 0,
           "longClickable": false,
           "scrollX": 0,
           "scrollY": 0,
           "isSelected": false,
           "isOpaque": true,
           "text": null,
           "textBackgroundColor": 1,
           "textColor": 1,
           "textLineBaselines": null,
           "textLineCharOffsets": null,
           "textSelectionEnd": -1,
           "textSelectionStart": -1,
```

```
          "textSize": 0,
          "textStyle": 0,
          "top": 0,
          "transformation": null,
          "visibility": 0,
          "webDomain": null,
          "width": 1080,
          "children": [
            {
              "accessibilityFocused": false,
              "activated": false,
              "alpha": 1,
              "assistBlocked": false,
              "autofillHints": [
                "username"
              ],
              "autofillOptions": null,
              "autofillType": 1,
              "autofillValue": null,
              "checkable": false,
              "checked": false,
              "className": "android.widget.EditText",
              "clickable": true,
              "contentDescription": null,
              "contextClickable": false,
              "elevation": 0,
              "enabled": true,
              "focusable": true,
              "focused": true,
              "height": 118,
              "hint": "Username",
              "htmlInfo": null,
              "id": 2131165193,
              "idEntry": "username",
              "idPackage": "com.commonsware.android.autofill.client",
              "idType": "id",
              "inputType": 97,
              "left": 21,
              "longClickable": true,
              "scrollX": 0,
              "scrollY": 0,
              "isSelected": false,
              "isOpaque": false,
              "text": "",
              "textBackgroundColor": 1,
              "textColor": 1,
              "textLineBaselines": null,
              "textLineCharOffsets": null,
```

```
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
      "top": 231,
      "transformation": null,
      "visibility": 0,
      "webDomain": null,
      "width": 1038,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": [
        "password"
      ],
      "autofillOptions": null,
      "autofillType": 1,
      "autofillValue": null,
      "checkable": false,
      "checked": false,
      "className": "android.widget.EditText",
      "clickable": true,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": true,
      "focused": false,
      "height": 118,
      "hint": "Passphrase",
      "htmlInfo": null,
      "id": 2131165194,
      "idEntry": "passphrase",
      "idPackage": "com.commonsware.android.autofill.client",
      "idType": "id",
      "inputType": 129,
      "left": 21,
      "longClickable": true,
      "scrollX": 0,
      "scrollY": 0,
      "isSelected": false,
      "isOpaque": false,
      "text": "",
      "textBackgroundColor": 1,
```

```
      "textColor": 1,
      "textLineBaselines": null,
      "textLineCharOffsets": null,
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
      "top": 370,
      "transformation": null,
      "visibility": 0,
      "webDomain": null,
      "width": 1038,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": [
        "creditCardNumber"
      ],
      "autofillOptions": null,
      "autofillType": 1,
      "autofillValue": null,
      "checkable": false,
      "checked": false,
      "className": "android.widget.EditText",
      "clickable": true,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": true,
      "focused": false,
      "height": 118,
      "hint": "Credit card number",
      "htmlInfo": null,
      "id": 2131165195,
      "idEntry": "cc",
      "idPackage": "com.commonsware.android.autofill.client",
      "idType": "id",
      "inputType": 97,
      "left": 21,
      "longClickable": true,
      "scrollX": 0,
      "scrollY": 0,
      "isSelected": false,
```

```
      "isOpaque": false,
      "text": "",
      "textBackgroundColor": 1,
      "textColor": 1,
      "textLineBaselines": null,
      "textLineCharOffsets": null,
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
      "top": 509,
      "transformation": null,
      "visibility": 0,
      "webDomain": null,
      "width": 1038,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": null,
      "autofillOptions": null,
      "autofillType": 0,
      "autofillValue": null,
      "checkable": false,
      "checked": false,
      "className": "android.widget.Button",
      "clickable": true,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": true,
      "focused": false,
      "height": 126,
      "hint": null,
      "htmlInfo": null,
      "id": 2131165196,
      "idEntry": "save",
      "idPackage": "com.commonsware.android.autofill.client",
      "idType": "id",
      "inputType": 0,
      "left": 828,
      "longClickable": false,
      "scrollX": 0,
      "scrollY": 0,
```

```
      "isSelected": false,
      "isOpaque": false,
      "text": "Save",
      "textBackgroundColor": 1,
      "textColor": 1,
      "textLineBaselines": null,
      "textLineCharOffsets": null,
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
      "top": 648,
      "transformation": null,
      "visibility": 0,
      "webDomain": null,
      "width": 231,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": null,
      "autofillOptions": null,
      "autofillType": 0,
      "autofillValue": null,
      "checkable": false,
      "checked": false,
      "className": "android.widget.TextView",
      "clickable": false,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": false,
      "focused": false,
      "height": 71,
      "hint": null,
      "htmlInfo": null,
      "id": -1,
      "idEntry": null,
      "idPackage": null,
      "idType": null,
      "inputType": 0,
      "left": 42,
      "longClickable": false,
      "scrollX": 0,
```

```
              "scrollY": 0,
              "isSelected": false,
              "isOpaque": false,
              "text": "",
              "textBackgroundColor": 1,
              "textColor": 1,
              "textLineBaselines": null,
              "textLineCharOffsets": null,
              "textSelectionEnd": -1,
              "textSelectionStart": -1,
              "textSize": 0,
              "textStyle": 0,
              "top": 101,
              "transformation": null,
              "visibility": 0,
              "webDomain": null,
              "width": 202,
              "children": []
            }
          ]
        }
      }
    ]
  }
]
}
```

Here is the data that the autofill service sees, from the same form, when it is time to save what the user filled in for later autofill reuse:

```
{
  "data": {
    "logDirName":
"\/storage\/emulated\/0\/Android\/data\/com.commonsware.android.autofill.logger\/cache\/autofilllogger_
  },
  "contexts": [
    {
      "windows": [
        {
          "displayId": 0,
          "height": 1920,
          "left": 0,
          "title":
"com.commonsware.android.autofill.client\/com.commonsware.android.autofill.client.MainActivity",
          "top": 0,
          "width": 1080,
          "root": {
```

```
          "accessibilityFocused": false,
          "activated": false,
          "alpha": 1,
          "assistBlocked": false,
          "autofillHints": null,
          "autofillOptions": null,
          "autofillType": 0,
          "autofillValue": null,
          "checkable": false,
          "checked": false,
          "className": "android.widget.FrameLayout",
          "clickable": false,
          "contentDescription": null,
          "contextClickable": false,
          "elevation": 0,
          "enabled": true,
          "focusable": false,
          "focused": false,
          "height": 1920,
          "hint": null,
          "htmlInfo": null,
          "id": -1,
          "idEntry": null,
          "idPackage": null,
          "idType": null,
          "inputType": 0,
          "left": 0,
          "longClickable": false,
          "scrollX": 0,
          "scrollY": 0,
          "isSelected": false,
          "isOpaque": true,
          "text": null,
          "textBackgroundColor": 1,
          "textColor": 1,
          "textLineBaselines": null,
          "textLineCharOffsets": null,
          "textSelectionEnd": -1,
          "textSelectionStart": -1,
          "textSize": 0,
          "textStyle": 0,
          "top": 0,
          "transformation": null,
          "visibility": 0,
          "webDomain": null,
          "width": 1080,
          "children": [
            {
```

```
              "accessibilityFocused": false,
              "activated": false,
              "alpha": 1,
              "assistBlocked": false,
              "autofillHints": [
                "username"
              ],
              "autofillOptions": null,
              "autofillType": 1,
              "checkable": false,
              "checked": false,
              "className": "android.widget.EditText",
              "clickable": true,
              "contentDescription": null,
              "contextClickable": false,
              "elevation": 0,
              "enabled": true,
              "focusable": true,
              "focused": true,
              "height": 118,
              "hint": "Username",
              "htmlInfo": null,
              "id": 2131165193,
              "idEntry": "username",
              "idPackage": "com.commonsware.android.autofill.client",
              "idType": "id",
              "inputType": 97,
              "left": 21,
              "longClickable": true,
              "scrollX": 0,
              "scrollY": 0,
              "isSelected": false,
              "isOpaque": false,
              "text": "foo",
              "textBackgroundColor": 1,
              "textColor": 1,
              "textLineBaselines": null,
              "textLineCharOffsets": null,
              "textSelectionEnd": -1,
              "textSelectionStart": -1,
              "textSize": 0,
              "textStyle": 0,
              "top": 231,
              "transformation": null,
              "visibility": 0,
              "webDomain": null,
              "width": 1038,
              "children": []
```

```
      },
      {
        "accessibilityFocused": false,
        "activated": false,
        "alpha": 1,
        "assistBlocked": false,
        "autofillHints": [
          "password"
        ],
        "autofillOptions": null,
        "autofillType": 1,
        "checkable": false,
        "checked": false,
        "className": "android.widget.EditText",
        "clickable": true,
        "contentDescription": null,
        "contextClickable": false,
        "elevation": 0,
        "enabled": true,
        "focusable": true,
        "focused": false,
        "height": 118,
        "hint": "Passphrase",
        "htmlInfo": null,
        "id": 2131165194,
        "idEntry": "passphrase",
        "idPackage": "com.commonsware.android.autofill.client",
        "idType": "id",
        "inputType": 129,
        "left": 21,
        "longClickable": true,
        "scrollX": 0,
        "scrollY": 0,
        "isSelected": false,
        "isOpaque": false,
        "text": "password",
        "textBackgroundColor": 1,
        "textColor": 1,
        "textLineBaselines": null,
        "textLineCharOffsets": null,
        "textSelectionEnd": -1,
        "textSelectionStart": -1,
        "textSize": 0,
        "textStyle": 0,
        "top": 370,
        "transformation": null,
        "visibility": 0,
        "webDomain": null,
```

```
      "width": 1038,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": [
        "creditCardNumber"
      ],
      "autofillOptions": null,
      "autofillType": 1,
      "checkable": false,
      "checked": false,
      "className": "android.widget.EditText",
      "clickable": true,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": true,
      "focused": false,
      "height": 118,
      "hint": "Credit card number",
      "htmlInfo": null,
      "id": 2131165195,
      "idEntry": "cc",
      "idPackage": "com.commonsware.android.autofill.client",
      "idType": "id",
      "inputType": 97,
      "left": 21,
      "longClickable": true,
      "scrollX": 0,
      "scrollY": 0,
      "isSelected": false,
      "isOpaque": false,
      "text": "123",
      "textBackgroundColor": 1,
      "textColor": 1,
      "textLineBaselines": null,
      "textLineCharOffsets": null,
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
      "top": 509,
      "transformation": null,
```

```
      "visibility": 0,
      "webDomain": null,
      "width": 1038,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": null,
      "autofillOptions": null,
      "autofillType": 0,
      "autofillValue": null,
      "checkable": false,
      "checked": false,
      "className": "android.widget.Button",
      "clickable": true,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": true,
      "focused": false,
      "height": 126,
      "hint": null,
      "htmlInfo": null,
      "id": 2131165196,
      "idEntry": "save",
      "idPackage": "com.commonsware.android.autofill.client",
      "idType": "id",
      "inputType": 0,
      "left": 828,
      "longClickable": false,
      "scrollX": 0,
      "scrollY": 0,
      "isSelected": false,
      "isOpaque": false,
      "text": "Save",
      "textBackgroundColor": 1,
      "textColor": 1,
      "textLineBaselines": null,
      "textLineCharOffsets": null,
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
      "top": 648,
```

```
      "transformation": null,
      "visibility": 0,
      "webDomain": null,
      "width": 231,
      "children": []
    },
    {
      "accessibilityFocused": false,
      "activated": false,
      "alpha": 1,
      "assistBlocked": false,
      "autofillHints": null,
      "autofillOptions": null,
      "autofillType": 0,
      "autofillValue": null,
      "checkable": false,
      "checked": false,
      "className": "android.widget.TextView",
      "clickable": false,
      "contentDescription": null,
      "contextClickable": false,
      "elevation": 0,
      "enabled": true,
      "focusable": false,
      "focused": false,
      "height": 71,
      "hint": null,
      "htmlInfo": null,
      "id": -1,
      "idEntry": null,
      "idPackage": null,
      "idType": null,
      "inputType": 0,
      "left": 42,
      "longClickable": false,
      "scrollX": 0,
      "scrollY": 0,
      "isSelected": false,
      "isOpaque": false,
      "text": "AFClient",
      "textBackgroundColor": 1,
      "textColor": 1,
      "textLineBaselines": null,
      "textLineCharOffsets": null,
      "textSelectionEnd": -1,
      "textSelectionStart": -1,
      "textSize": 0,
      "textStyle": 0,
```

```
                "top": 101,
                "transformation": null,
                "visibility": 0,
                "webDomain": null,
                "width": 202,
                "children": []
              }
            ]
          }
        }
      ]
    }
  ]
}
```

Roughly speaking, you get all the same information that an accessibility service might. Most of this information is likely to be useless to an autofill service, but a few elements, such as `idEntry` (the widget ID) and `text` (the contents of a `TextView` or things inheriting from `TextView`) will be of importance.

Blocking Autofill as a User

The O Developer Preview 3 ships with an "Autofill with Google" autofill service, pre-installed and pre-enabled. Presumably, many devices that ship with Android 8.0 will ship with an autofill service pre-installed, whether that is one from Google, from the device manufacturer, or from another party.

If you, as a user, do not trust the autofill service, you can disable it or switch to a different one via the Settings app. Specifically, Settings > System > Languages & input > Advanced has an "Autofill service" preference:

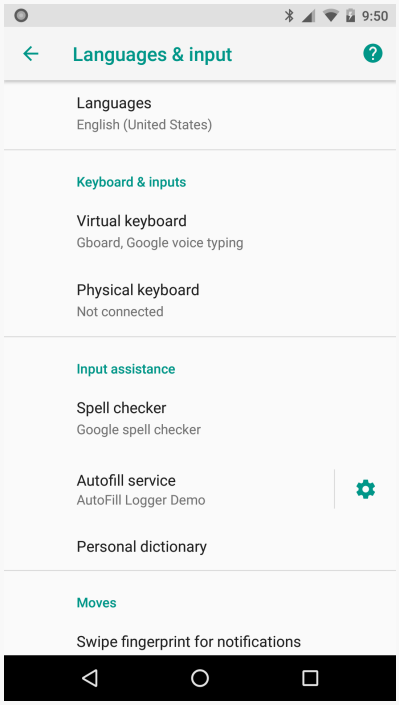


Figure 703: Advanced "Languages & input" in Settings, Showing "Autofill service"

Tapping the gear icon next to the configured autofill service (if there is one) will allow you to configure the service. Tapping the rest of the list row brings up a dialog for you to choose an autofill service from the available candidates, or "None" to opt out of autofill entirely:

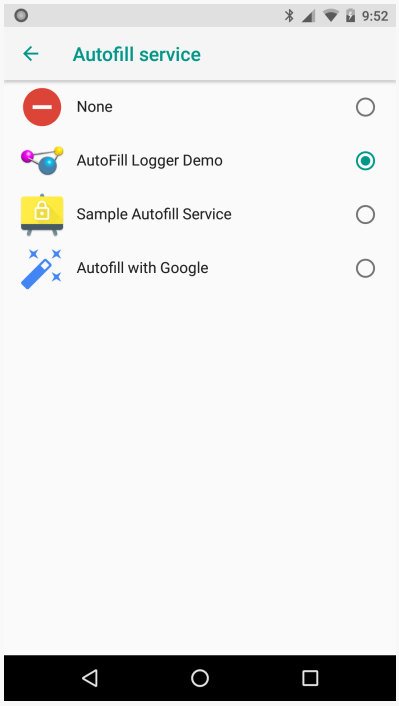


Figure 704: "Autofill service" Selection in Settings

Supporting Autofill with Standard Widgets

The documentation suggests that apps do not need to be modified to work with the Autofill Framework. That is incorrect, at least in general. Layouts do need some modification to work with autofill services, to help those autofill services identify the roles of those widgets and how those roles tie into saved data that could be autofilled-in.

The thing to bear in mind about autofill is that we have three parties to the content negotiation:

- Your app
- The Android Autofill Framework
- The autofill service implementation

This requires some common ground, as otherwise the autofill service will not know what is and is not to be considered candidates for autofill.

Identifying Roles via Hints

The primary thing that you need to do is to identify the roles of various widgets in your form. After all, a field containing a user ID might be called `userid`, `userId`, `uid`, `username`, `user_name`, `login`, or anything else. An autofill service needs to know that your `EditText` with an `android:id` of `@+id/snicklefritz` represents a user ID, for example.

To do this, use `android:autofillHints` in a layout resource, or `setAutofillHints()` from Java. These indicate, for the particular `View` you are configuring what role or role(s) the `View` holds within the world of autofill.

Autofill hint values are strings. Typically, a given widget only needs one hint, and so you can set the hints to that particular string. If, for whatever reason, a widget qualifies for more than one hint, use a comma-delimited list as the value for `android:autofillHints`, or pass multiple strings to `setAutofillHints()` via varargs.

The roster of possible hints, by name and Java `View` constant, are:

Value	`View` Constant
creditCardExpirationDate	AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DATE
creditCardExpirationDay	AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_DAY
creditCardExpirationMonth	AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_MONTH
creditCardExpirationYear	AUTOFILL_HINT_CREDIT_CARD_EXPIRATION_YEAR
creditCardNumber	AUTOFILL_HINT_CREDIT_CARD_NUMBER
creditCardSecurityCode	AUTOFILL_HINT_CREDIT_CARD_SECURITY_CODE
emailAddress	AUTOFILL_HINT_EMAIL_ADDRESS
name	AUTOFILL_HINT_NAME
password	AUTOFILL_HINT_PASSWORD
phone	AUTOFILL_HINT_PHONE
postalAddress	AUTOFILL_HINT_POSTAL_ADDRESS
postalCode	AUTOFILL_HINT_POSTAL_CODE
username	AUTOFILL_HINT_USERNAME

Here, `name` is for a real name (e.g., “Mark Murphy”), whereas `username` is for something more like a login value (e.g., “commonsguy”).

Since these are strings, and since the Android Autofill Framework (probably) does not care about the specific values, it is theoretically possible for autofill services to document additional hints that they honor (e.g., `birthDate`).

Indicating Importance

Independent from the hint system is `android:importantForAutofill` and the corresponding `setImportantForAutofill()` method. These indicate whether you, as the developer of the UI, think that certain widgets should or should not be considered for autofill purposes. For example, you might have a form that a user might fill out repeatedly and might want autofill capability, but some fields do not fit specific roles governed by the hint system, and so the autofill service might ignore them by default.

The default importance is `auto`, meaning that the autofill service will make its own guess, probably based on whether there is an autofill hint or not. This maps to `IMPORTANT_FOR_AUTOFILL_AUTO` in `setImportantForAutofill()`.

There are four alternatives:

XML Attribute Value	Java Constant	Role
`no`	`IMPORTANT_FOR_AUTOFILL_NO`	widget should be ignored from an autofill standpoint
`noExcludeDescendants`	`IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS`	widget and all its children should be ignored
`yes`	`IMPORTANT_FOR_AUTOFILL_YES`	widget should always be considered for autofill
`yesExcludeDescendants`	`IMPORTANT_FOR_AUTOFILL_YES_EXCLUDE_DESCENDANTS`	widget and all its children should be considered

For example:

- To block autofill from seeing the value of a sensitive widget, use `android:importantForAutofill="no"` or `setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_NO)` on the affected `View`
- To block autofill for an entire activity, use this in `onCreate()` of the activity:

```
getWindow()
  .getDecorView()
  .setImportantForAutofill(View.IMPORTANT_FOR_AUTOFILL_NO_EXCLUDE_DESCENDANTS);
```

Supporting Autofill with Custom Widgets

Other than having identifiable roles via hints, standard widgets are handled automatically via the Autofill Framework. However, the framework has no idea how to handle custom views of your own design, particularly those that use low-level drawing APIs like `Canvas` to render their contents.

The O Developer Preview documentation has some instructions, for developers of custom views to follow, if and where those views should be participating in autofill.

Dealing with Dynamic Changes

Sometimes, the form content changes significantly on the fly as the user is interacting with it. For example, you might dynamically add some widgets based on user input (e.g., tapping a "+" icon to add a row to a table). Hence, there may be cases when you need to tell the Autofill Framework more explicitly when it should do (or re-do) its work.

There is an `AutofillManager` system service that is your gateway to the Autofill Framework. You can obtain an instance from `getSystemService(AutofillManager.class)` called on a convenient `Context`, such as your activity.

Then:

- If you need to tell the Autofill Framework to discard any existing autofill logic applied to the current form (e.g., the user did something to reset the form contents), call `cancel()` on the `AutofillManager`
- If you want to ask the Autofill Framework specifically to re-apply the autofill logic for the current form, call `requestAutofill()` on the `AutofillManager`

- If you want to do something differently depending upon whether or not autofill is enabled, call `isEnabled()` on the `AutofillManager`
- If you want the Autofill Framework to complete its processing of the current form contents, without waiting for the activity to be destroyed, call `commit()` on the `AutofillManager`

So, for example, if you want to bail out of your activity right away if autofill is enabled — for example, you are concerned that you cannot adequately defend the privacy of your users — you could have the following code snippet in `onCreate()` of the activity:

```
if (getSystemService(android.view.autofill.AutofillManager.class).isEnabled()) {
  Toast.makeText(this, "Ick!", Toast.LENGTH_LONG).show();
  finish();
}
```

(where you would replace the `Toast` with something else to explain the situation to the user)

Or, if you wanted to ensure that the contents of your form were not submitted to the autofill service — again, with privacy in mind — you can `cancel()` the autofill operation in `onBackPressed()`, or perhaps in other lifecycle methods:

```
@Override
public void onBackPressed() {
  getSystemService(android.view.autofill.AutofillManager.class).cancel();

  super.onBackPressed();
}
```

Security Requirements of Autofill Services

> Of course, a poorly written autofill provider can leak sensitive data.

That quote is from a comment on a security issue regarding a security flaw in Android 8.0's autofill implementation. While those links may not work for you, as the issue is still locked at the time this was written, it contains a long chain of comments about this flaw and how Google expects the Android ecosystem to deal with the flaw.

First, the Flaw

Autofill will fill in widgets that the user cannot see, because:

- The widget is marked as invisible (`android:visibility="invisible"`)
- The widget has no size (width and height of `0dp`) or is impossibly tiny (width and height of `1dp`)
- The widget has negative margins that cause it to display off-screen
- The widget is behind some other opaque widget (on the Z axis), and so the widget cannot be seen

There may be other similar scenarios. For example some discussion on the issue suggests that autofill will supply data for a widget that does not exist at all in the view hierarchy, but rather is faked by the activity as part of how it populates the `ViewStructure` of information that goes to the autofill service.

Given this flaw, a malicious activity could obtain data that the user does not realize will go to that activity. For example, the activity might have a field for the user to confirm their postal code and have hidden widgets that collect other data, such as the rest of the address, credit card details, usernames/passphrases, etc.

This flaw can be demonstrated using Google's own sample autofill service, though in the future that sample should be updated to show a more secure implementation.

A copy of the flawed autofill service sample, along with a client demonstrating how malicious activities can obtain data covertly from the autofill service, can be found in this GitHub repo. Instructions for reproducing the problem can be found there as well.

Google's Response

Google's engineers on the security issue admit that the flaw exists and that it would be difficult, if not impossible, to defend against malicious activities.

As such, Google is taking the approach of dumping this issue on the laps of the developers of autofill services. That is the context for the quote at the top; Google believes that well-crafted autofill services will resolve this issue. In the issue, Google has stated that their own autofill service implementation that ships pre-installed and pre-enabled on Android 8.0 will be well-crafted.

There are two problems:

1. At present, there is little documentation or sample code on what a well-crafted autofill service looks and works like, so many other developers looking to implement autofill services are left in the dark
2. Their advice will not resolve the problem completely, though it will reduce the scope

What Secure Autofill Services Need To Do

Unfortunately, the documentation for writing autofill services is limited and does not address very well the issue of how to write a secure autofill service.

Instead, Google "work[s] closely with most major autofill providers", focusing on that instead of documentation, sample code, and public test suites. This approach is difficult to defend, given that there may be entire markets for which Google's efforts have no impact (e.g., China, if none of the "major autofill providers" distribute their apps there).

So, what is it that Google is expecting autofill service developers to do?

Partition the Dataset

The one bit of advice that Google has published publicly (as of the time this was written) was to partition the dataset. This involves two steps:

1. Cluster the possible requested autofill hints into partitions (e.g., address/phone versus credit card versus username/password)
2. Only hand back the data for one partition at a time, based on what widget has the focus at the time the autofill request is made of the service

So, for example, suppose the form has fields for address, phone number, and the typical batch of credit card details (number, name, expiration date, etc.) A poorly-crafted autofill service provides autofill data for all of that when Android asks for it... despite the fact that this is what the API would appear to expect the autofill service to do. Instead, you need to:

- Identify which widget has the focus
- Determine which partition (address/phone or credit card) that widget belongs to
- Only supply autofill data for that partition on this request

Later on, if the user puts the focus on a field that is eligible for autofill but for which you did not provide the data previously, you will get a second chance at the same form, where you repeat the process, only supplying autofill data for the partition associated with the now-focused partition.

See the "Data Partitioning" section in the `AutofillService` JavaDocs for complete details.

Only Give Data Back to the App That Supplied It

Another bit of advice: only give data back to the app that supplied it, for anything that you or the user might consider to be sensitive.

On the Web, some autofill works across all domains. Data that you fill into a form on one Web site can be captured by the browser and offered up as candidate responses on other sites, including ones that you have never visited before. If the user fills in an email address for Site A, that email address is available to supply via autofill to Site B.

The equivalent behavior with Android's autofill would be for the autofill service to have a single pool of data that could be autofilled-in for any app. If the user fills in an email address for App A, that email address is available to supply via autofill to App B.

Google feels that this is fine, so long as the data is not something that the user would consider to be sensitive. In particular, they recommend that usernames and passwords be locked to an individual app and not pooled, which makes perfect sense. For such sensitive content, an autofill service must only provide data back to the specific app that supplied it, via a combination of the `applicationId` ("package") of the app and the public signing key of that app:

> An autofill provider must verify the *package+signature* to prevent phishing - this guarantees that even if a sideloaded app uses another app's package name and mimics its UI cannot get access to sensitive data associated with it.

At the present time, this "only give data back to the app that supplied it" requirement is undocumented, outside of the aforementioned security issue. According to the developers, it will be documented sometime in the future.

Hint to the User What Data Is Being Autofilled In

The autofill service API has support for the service to require authentication before actually providing the autofill data. In a nutshell, the service creates a `PendingIntent` pointing to the authentication activity associated with the autofill service and includes that `PendingIntent` in its response to an autofill request. If the user elects to proceed, the `PendingIntent` is used to display the authentication activity. If the user successfully authenticates, the user is returned to the activity they had been on previously, but now the secured autofill data should now be available for use.

For anything that is deemed sensitive, Google not only recommends that autofill services use this authentication flow, but that the authentication activity display details of exactly what data will be released. This way, for whatever autofill request triggered the authentication, the user will be informed about all fields that would be filled in... including those that the user cannot see.

This requirement is also undocumented at this time.

What Google's Advice Does Not Solve

There are gaps in Google's advice.

First, there is no real enforcement that any autofill service follow the advice. This might be checked manually for the Play Store:

> However, we'll have a manual procedure in place to verify such compliance for apps that want to be list as an Autofill service option in the "Add Autofill Service" Play Store page that is launched from Settings.

How well this works, and whether any other app distribution channel will implement similar checks, remains to be seen. Similarly, whether any security firms or other independent parties offer some sort of compliance check remains to be seen.

Second, the "only give data back to the app that supplied it" rule will fail for apps for whom the form and the data come from outside the app... such as for Web browsers. There, the app is the *browser*, not the Web site. The autofill service has no obvious means of distinguishing one site from another, or even that there is a concept of "site" that needs to be taken into account.

Third, what the *user* considers to be sensitive data, what *Google* considers to be sensitive data, and what *autofill service developers* consider to be sensitive may differ. For example, the user might be rather concerned about their address being made available. But if the autofill service developers do not consider that to be sensitive data, Google's recommendations may be ignored. In particular, if you implement the partitioning, but do not limit data sharing between apps, a malicious activity can still steal data within a single partition, such as obtaining all the credit card details when just asking visibly for the expiration year.

Fourth, the hint-when-authenticating approach helps, but a malicious activity can defeat simple implementations, by requesting multiple partitions and setting the focus such that an innocuous partition is requested first. For example, the malicious activity's form might have visible fields for email address and postal code, followed by invisible fields for the rest of the address. Putting the email address field first and giving it the focus will cause any authentication to happen for the email partition. If the authentication activity only hints to the user about fields that will be autofilled in from that one partition, the user will only be told about the visible email address field. Authentication should not be required when the user advances to the postal code field, as the user just authenticated, and so the user might not receive the hint about the hidden address fields after the postal code field.

The Data Binding Framework

To quote Rudyard Kipling:

> East is East and West is West, and never the twain shall meet

In many programming environments, including classical Android development, one could paraphrase Kipling as "models are models and views are views, and never the twain shall meet, except by means of some controller or presenter or something". The result is a fair amount of code that populates views with model-supplied data and updates those models as the user alters the data in the views (e.g., types something in an `EditText` widget).

Data binding, in general, refers to frameworks or libraries designed to help simplify some of this data migration, where the definitions of the models and views can be used to automatically "bind" them without as much custom controller- or presenter-style logic.

Interest in data binding spiked in 2015, when Google released the first beta editions of data binding support via Android Studio, the Android Gradle Plugin, and a new data-binding support library.

This chapter explores Google's data binding support and how to use it to simplify your Android app development.

Prerequisites

This chapter requires that you have read the core chapters of this book. In particular, the sample apps are based off of samples from the chapter on Internet access. Also, some samples use `RecyclerView`.

The What, Now?

In that chapter on Internet access, we examined a few variations of a sample app that retrieved the latest `android` questions from Stack Overflow and displayed them in a `ListView`. Our `QuestionsFragment` had an `ItemsAdapter` for populating the `ListView`, complete with a typical `getView()` implementation:

```
    setListAdapter(new ItemsAdapter(questions.items));
  }

  class ItemsAdapter extends ArrayAdapter<Item> {
    ItemsAdapter(List<Item> items) {
      super(getActivity(), R.layout.row, R.id.title, items);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      Item item=getItem(position);
      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      Picasso.with(getActivity()).load(item.owner.profileImage)
             .fit().centerCrop()
             .placeholder(R.drawable.owner_placeholder)
             .error(R.drawable.owner_error).into(icon);

      TextView title=(TextView)row.findViewById(R.id.title);

      title.setText(Html.fromHtml(getItem(position).title));
```

(from HTTP/Picasso/app/src/main/java/com/commonsware/android/picasso/QuestionsFragment.java)

Some parts of `getView()` are clearly distinct for this application, notably using Picasso to download the question asker's avatar and using `Html.fromHtml()` to handle HTML-style entities in the title.

However, the general process used here in `getView()` is fairly rote:

- Get the widget out of the row
- Stuff the data into the widget for the row
- Do the above for each widget needing to be updated as part of binding data to the row (a.k.a., "lather, rinse, repeat")

Data binding, as a general technique, aims to reduce that rote coding by declaratively telling a framework how to pull data from model objects (e.g., instances of `Item`) and pour that data into widgets (e.g., `ImageView` and `TextView`).

The Basic Steps

With that in mind, let's examine what it takes to convert this sample over to using Google's data binding system.

The code samples shown in this section come from the `DataBinding/Basic` sample project.

Setting Up the Toolchain

Data binding only really works well with up-to-date versions of Android Studio (1.3 or higher) and the Android Gradle Plugin (1.5.0 or higher recommended).

The data binding system consists of two pieces: another plugin for Gradle, and a library that gets bundled with our app. However, we do not need to set those up manually. Instead, we simply tell the Android Gradle Plugin that we want data binding, and it adds the requisite plugin and library for us.

All we need is a small `dataBinding` closure, where we set the `enabled` property to `true`:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
}

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
    }
```

```
    dataBinding {
        enabled = true
    }
}
```

(from DataBinding/Basic/app/build.gradle)

Once you do this, future times that you open this project in Android Studio may result in you getting a "Source folders generated at incorrect location" message:

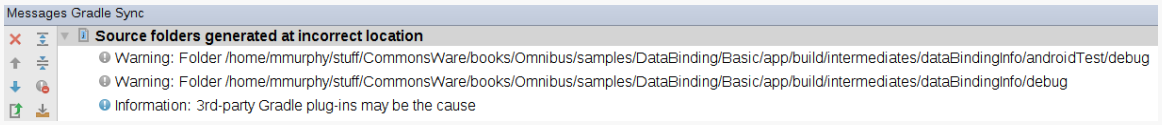


Figure 705: Data Binding Gradle Sync Message

This is due to a bug that, in the fullness of time, may get fixed. However, the messages appear to be benign, and they should not cause any problems with your app.

Augmenting the Layout… and the Model

The real fun begins with the layout for our `ListView` row. The original edition of this layout resource was a typical `LinearLayout` with an `ImageView` and `TextView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="@dimen/icon"
    android:layout_height="@dimen/icon"
    android:layout_gravity="center_vertical"
    android:contentDescription="@string/icon"
    android:padding="8dip"/>

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="20sp"
```

```
    android:layout_gravity="left|center_vertical"/>

</LinearLayout>
```

(from HTTP/Picasso/app/src/main/res/layout/row.xml)

We need to make some changes to that in order to leverage data binding:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">

  <data>

    <variable
      name="item"
      type="com.commonsware.android.databind.basic.Item"/>
  </data>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="@dimen/icon"
      android:layout_height="@dimen/icon"
      android:layout_gravity="center_vertical"
      android:contentDescription="@string/icon"
      android:padding="8dip"/>

    <TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical"
      android:text="@{item.title}"
      android:textSize="20sp"/>

  </LinearLayout>
</layout>
```

(from DataBinding/Basic/app/src/main/res/layout/row.xml)

First, the entire resource file gets wrapped in a `<layout>` element, on which we can place the `android` namespace declaration.

That <layout> element then has two children. The second child is our LinearLayout, representing the root View or ViewGroup for the resource. The first child is a <data> element, and that is where we configure how data binding should proceed when this layout resource gets used. Specifically, the <variable> element indicates that we want to bind data from an Item object into widgets defined in this layout.

Then, if you look at the TextView, you will see that it now has an android:text attribute that the original layout resource lacked. More importantly, the value for android:text is unusual: @{item.title}. The @{} syntax indicates that rather than interpreting the value as a plain string, or even a reference to a string resource, that the value is really an expression, in a data binding expression language, that should be computed at runtime to get the value to assign to the text of the TextView.

In this case, the expression is item.value. item is the name given to the Item object in the <variable> element. Any place where we want to pull data from that Item object, we can use dot notation to reference things on the item expression language variable.

item.value means "get the value from the item". At runtime, the data binding library will attempt to get this value either from a public getter method (getValue()) or a public field (value) on the Item class. The original project had a value field, but it was not public, so the revised project marks the Item fields as public, so we can use them in data binding:

```
package com.commonsware.android.databind.basic;

public class Item {
  public String title;
  public Owner owner;
  public String link;

  @Override
  public String toString() {
    return(title);
  }
}
```

(from DataBinding/Basic/app/src/main/java/com/commonsware/android/databind/basic/Item.java)

As we will see in this chapter, the expression language used here is much more complex than simply referencing JavaBean-style properties on objects, but for now, this will suffice.

Applying the Binding

The layout configures one side of the binding: pulling data into widgets. We still need to do some work to configure the other side of the binding: supplying the source of that data. In the case of this example, we need to provide the `Item` object for this layout resource.

That is handled via some modifications to the `getView()` method of the `ItemsAdapter` from its original version:

```
  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    RowBinding rowBinding=
      DataBindingUtil.getBinding(convertView);

    if (rowBinding==null) {
      rowBinding=
        RowBinding.inflate(getActivity().getLayoutInflater(),
          parent, false);
    }

    Item item=getItem(position);
    ImageView icon=rowBinding.icon;

    rowBinding.setItem(item);

    Picasso.with(getActivity()).load(item.owner.profileImage)
           .fit().centerCrop()
           .placeholder(R.drawable.owner_placeholder)
           .error(R.drawable.owner_error).into(icon);

    return(rowBinding.getRoot());
  }
```

(from DataBinding/Basic/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

There are four changes here: we create the binding, provide the model (`Item`) to the binding, retrieve other widgets from the binding, and retrieve the root view of the layout.

Creating the Binding

When we use `<layout>` in a layout resource and set up the layout side of the data binding system, the build system code-generates a Java class associated with that

layout file. The class name is derived from the layout name, where `names_like_this` get converted into `NamesLikeThis` and have `Binding` appended. So, since our layout resource was `row.xml`, we get `RowBinding`. This is code-generated into a `databinding` Java sub-package of the package name from the manifest. Hence, the fully-qualified import statement for this class is:

```
import com.commonsware.android.databind.basic.databinding.RowBinding;
```

This is a subclass of `ViewDataBinding`, supplied by the `databinding` library that is added to your project by enabling data binding in your `build.gradle` file.

Creating an instance of the binding also inflates the associated layout. Your binding class has a number of factory methods for inflating the layout and creating the binding. These mirror other methods that you have used elsewhere:

- `setContentView()`, taking an `Activity` and the layout resource ID as parameters, inflates the layout, passes the result to `setContentView()` on the `Activity`, and creates the binding
- `inflate()`, with a variety of parameter list options, just inflates the layout using a `LayoutInflater`, and creates the binding

Here, we use the three-parameter flavor of `inflate()`, which takes a `LayoutInflater` (obtained from the hosting activity), the parent container, and `false`. This mirrors the `inflate()` one would use on `LayoutInflater` itself, except that it also gives us our binding.

Of course, this is a `ListView`, and so we have to deal with the possibility that rows get recycled. The `DataBindingUtil` class has a `getBinding()` method that returns the binding for a given root view from the inflated layout — in this case, our `convertView`. So, we try to get the existing binding first, then fall back to inflating a new one if and only if that is necessary. Since `getBinding()` properly handles `null` values for `convertView`, we do not need to check for `null` ourselves explicitly.

Pouring the Model into the Binding

The generated binding class will have setters for each `<variable>` in our `<data>` element in the layout. Setter names are generated from the variable names using standard JavaBean conventions, so our `item` variable becomes `setItem()`. When we call `setItem()`, the data binding system will use that `Item` object to populate our `TextView`, applying the binding expression from our `android:text` attribute.

Retrieving Widgets from the Binding

However, we did not do anything related to data binding for the `ImageView` widget in the layout (though we will, later in this chapter). Hence, we still need to manage that manually, getting Picasso to fetch the avatar asynchronously and put it in the `ImageView`.

However, that implies that we *have* the `ImageView`. Normally, we would call `findViewById()` on the inflated layout's root `View` to obtain that.

However, our binding class has code-generated `public` fields on it for each widget in the layout resource that has an `android:id` value (at least for `@id/...` and `@+id/...` values). Our `ImageView` has an `android:id` value of `@+id/icon`, and so the `RowBinding` class has an `icon` field that holds our `ImageView`. We can simply reference it, rather than doing the `findViewById()` lookup ourselves.

Getting the Actual View

Since `getView()` is supposed to return the inflated layout's root view, we need some way to get that from the binding. Fortunately, `ViewDataBinding` has a `getRoot()` method that our generated class inherits, so we can just call that to get the value to return from `getView()`.

Results

Visually, this app is the same as before (though this version uses `Theme.Material` on compatible devices). Functionally, the app is the same as before. And, from a code complexity standpoint, the app is probably *worse* than before, as we went through a lot of work just to avoid calling `findViewById()` a couple of times and `setText()` once.

Hence, while the data binding system is nice, it really only adds value to larger projects, particularly those with complex layouts. By the end of this chapter, you should have a better sense for when data binding is useful and when it is overkill.

The Extended Layout Resource

As we saw in the preceding example, much of the knowledge that we impart into our app to power the data binding comes in the form of an extended layout resource syntax. The last child of the root `<layout>` element is what our layout resources used

to hold: the `View` or `ViewGroup` at the root of the view hierarchy of this layout. Other children of `<layout>` configure the data binding behavior (and perhaps other features in the future).

With that in mind, let's explore a bit more about what you can do with elements in the `<layout>`.

Imports and Statics

The preceding example lost one feature with respect to the sample app that served as its starting point: handling HTML in titles. While Stack Overflow does not serve HTML tags in question titles, it *does* serve HTML entities in question titles. A question title of "Foo & Bar" would come to us in the JSON as "Foo & Bar". The examples in the chapter on Internet access handle that via `Html.fromHtml()`. However, we do not have that in our data binding.

One way to address this is to add a getter-style method to `Item` that returns the title after passing through `Html.fromHtml()`. For example, we could have a `getInterpretedTitle()` or `getTitleWithEntitiesFixed()` or `getTitleAfterHavingRunItThroughHtmlFromHtml()`. We would then refer to that method in our `android:text` expression (e.g., `@{item.interpretedTitle}`).

However, this blurs the line dividing the responsibilities of model objects and the UI layer. The model itself does not care that the title has HTML entities in it, and some ways of using that model data (e.g., displaying in a `WebView`) might specifically *need* those HTML entities. The fact that we need to convert those HTML entities is a UI responsibility, because the UI chose to use a `TextView`, which does not handle those entities automatically.

A fairly easy way to get our `Html.fromHtml()` logic back in would be to apply it in the layout resource itself. It would be cool if we could have our expression be `@{Html.fromHtml(item.title)}`, for example.

The good news is: that is eminently possible.

However, if you just used that syntax without other changes, the data binding framework would complain that it does not know what `Html` is. In effect, we need to teach the layout resource where to import `Html` from.

To do that, we need to add `<import type="android.text.Html"/>` into the `<data>` element of our layout resource. Now, the generated code will contain that `import` statement and our references to `Html` will resolve.

You can see that in the DataBinding/Static sample project. This is a clone of `DataBinding/Basic` with the two changes (expression and `<import>` applied), giving us the following layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">

  <data>

    <import type="android.text.Html"/>

    <variable
      name="item"
      type="com.commonsware.android.databind.basic.Item"/>
  </data>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="@dimen/icon"
      android:layout_height="@dimen/icon"
      android:layout_gravity="center_vertical"
      android:contentDescription="@string/icon"
      android:padding="8dip"/>

    <TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical"
      android:text="@{Html.fromHtml(item.title)}"
      android:textSize="20sp"/>

  </LinearLayout>
</layout>
```

(from DataBinding/Static/app/src/main/res/layout/row.xml)

If you run this version of the app, and it so happens that there is a Stack Overflow question with an HTML entity in its title among the recent questions, you will see that entity show up properly in the `ListView`. On the other hand, if you run the previous sample, the HTML entity will show up in HTML source form (e.g., `&` instead of `&`).

The rules for imports here are reminiscent of those of regular Java:

- Do not have conflicting imports (e.g., `android.view.Menu` and `com.myrestaurant.Menu`)
- Do not try to import classes that are automatically imported (e.g., `java.lang.String`)

Variables

As we saw in the preceding samples, you can have `<variable>` elements representing objects that can be referenced by binding expressions.

The `type` attribute for the `<variable>` element can be:

- a fully-qualified class name, as seen in the `item` variable from the examples
- the name of a class that you added via an `<import>` element
- the name of any class automatically imported into all Java classes (e.g., `Integer`)

So, for example, instead of:

```
<data>
  <variable
    name="item"
    type="com.commonsware.android.databind.basic.Item"/>
</data>
```

we could have:

```
<data>
  <import type="com.commonsware.android.databind.basic.Item"/>

  <variable
    name="item"
    type="Item"/>
</data>
```

If you have different versions of the same layout in different resource sets for different configurations (e.g., `res/layout/` and `res/layout-land/`), your `<layout>` element needs to be compatible between them. This particularly holds true with respect to variables. If you define a variable `foo` as a `String` in one version of the resource, you cannot define `foo` to be a `Restaurant` in another version of the resource. There is one binding class created for each layout resource, spanning all of the different versions of that resource, and that class cannot have two separate, conflicting definitions for the same variable.

The Binding Expression Language

To a basic approximation, the binding expression language that you can use in layout resources works just like its Java counterpart. If you can include it in a Java expression, you can include it in a binding expression. This not only covers your typical mathematical, logical, and string concatenation operations, but also:

- Casts
- Using parentheses for grouping (e.g., `@{((Location)(restaurant.location)).latitude}`)
- Calling methods, both on objects in the expression and `static` methods on imported classes
- Accessing fields by name, both on objects and on `static` classes that you have imported
- Accessing array contents using square-bracket notation, including using other variables as the index (e.g., `@{movie.actor[index].fullName}`)
- Using the ternary operator for inline `if`-style branching (e.g., `@{movie.isNew ? View.VISIBLE : View.GONE}`)

Stuff You Won't Find in Java

The expression language contains a few conveniences that go beyond what you will see in standard Java.

One of these has already been mentioned: JavaBean-style accessor usage. So, `foo.bar` will try to find a field named `bar` on the `foo` object. If that is not found, `foo.bar` will try to find a `getBar()` method on the `foo` object. This allows the model object to decide whether or not to expose the data via a field or getter method; the binding expression works with either.

If you have a variable that is a `Map`, you can use square-bracket notation to access the map by key, instead of having to call `get()`.

If you try accessing a field or calling a method on `null`, you normally would get a `NullPointerException`. The expression evaluator tries to mitigate that:

- If the field or method is designed to return some primitive, the result of accessing the field or calling the method on `null` returns whatever the default primitive value is (e.g., `int` and `long` values are `0`)
- Otherwise, if the field or method returns some object, the result of accessing the field or calling the method on `null` returns `null`

Another way of working with `null` values is the `??` "null coalescing operator". In the expression `foo ?? bar`, the result is:

- `foo`, if `foo` is not `null`
- `bar`, if `foo` is `null`

This is useful when you want to replace some optional value with a default when the optional value is `null`. For example, you might use `sub.expirationDate ?? @string/not_yet_subscribed` to either show the expiration date of some subscription, or pull in the value of a string resource to use if there is no expiration date.

That example demonstrates yet another feature of the expression language: references to resources. In general, you reference them just as you would without the data binding system. So, these are equivalent:

- `android:text="@string/foo"`
- `android:text="@{@string/foo}"`

Of course, the power comes in when using those resources in actual expressions, such as using a `boolean` resource with the ternary operator (e.g., `@{@boolean/i_can_haz_foo ? foo : bar}`).

Note that a few resource types use different names in the binding expressions, as the expression evaluator needs to know the data type. So, for example, you normally reference array resources as simply `@array/name`. In binding expressions, you replace `@array` with a different symbol to indicate the type, such as `@stringArray` or `@intArray`.

Caveats

Of course, if all of this were simple, it wouldn't be Android...

Handling String Literals

Numeric literals and `null` can be used in expressions easily enough. String literals get interesting, as the standard Java `"` quotation system runs afoul of the default XML `"` quotation system for attribute values. Your options are:

- Use single quotes for the XML attribute, so you can use double quotes for the string literal (e.g., `android:text='@{foo["bar"]}'`)
- Use backticks for the string delimiter instead of double quotes (e.g., `android:text="@{foo[\bar`]}"`)
- Use HTML-style `"` entities for the string delimiter (e.g., `android:text="@{foo["bar"]}"`)

Of the three, the latter one is your worst choice, in terms of readability.

Watch Out For Mis-Interpreted Integers

Suppose that you want to have the `android:text` attribute of a `TextView` hold a numeric value, pulled from a variable. You might try using something like `android:text="@{question.score}"`, where `score` is an `int`.

When you try it, you will crash at runtime, with an error indicating that there is no resource with the ID of some hex value, where that hex value *happens* to be your `score`.

That is because `android:text` supports strings *or string resources*. The integer value for `score` will be interpreted as a reference to a string resource, not converted into a string itself.

You then might try `android:text="@{question.score.toString()}"`. That fails to compile, if `score` is an `int`, as Java primitives do not support methods, let alone `toString()`.

The right solution is to use `static` methods on `Integer` to convert the `int` into a string: `android:text="@{Integer.toString(question.score)}"`

Other Caveats

Because this stuff appears in plain XML, you will need to escape any < or > signs used in the expressions as `<` and `>`, respectively, which is aggravating.

You cannot use the `new` operator to create objects. However, you are welcome to call methods that happen to create new objects. So, in a pinch, create yourself a factory method somewhere to create the object that you were trying to instantiate via `new`. All things considered, though, the more object instantiation you do in layout binding, the slower that binding can become, particularly for oft-inflated layouts like rows in a rapidly-scrolling list.

You do not have access to `this` or `super`, as these would be with reference to the generated binding class itself.

Observables and Updating the Binding

Variables, and the fields or method results that you access on them, can populate `View` properties, as we have seen so far in this chapter. This is interesting, but it may not "move the needle" for you in terms of adopting data binding. While there may be some minor code maintenance benefit, it hardly seems worth it.

Where data binding really shines, though, is when the variables, and the fields or method results that you access on them, are observable objects (i.e., ones implementing `android.databinding.Observable`). Then, not only do the expressions update your `View` properties when the layout resources are inflated, **but also when the data changes**. If you have observable models, simply updating those model objects automatically propagates those changes to any live `View` objects looking at those models.

For example, suppose that you are writing a to-do sort of checklist. The user can tap a `CheckBox` widget to indicate that the particular task is completed, and at that point you want to change the rendering of the task overall in its `RecyclerView` row in addition to updating the model object representing the task. Since the `CheckBox` is part of that same row, bound to the model for the row, handling both the UI updates and the model updates in the same `OnClickListener` may be easy. However, what happens if you do not want to update the rendering until the model change has been saved to the database or the network? Now, some arbitrary number of milliseconds *after* `OnClickListener` returns, you need to update some row of the `RecyclerView`... if there happens to be a row pointing at this model object. After all,

the user might have scrolled, or even left this `RecyclerView` entirely, in which case the original row should not be changed.

The obvious tradeoff is defining your model objects to use `Observable`. The less-obvious tradeoff is in reorganizing your code to have *durable model objects*, where operations like Web service calls update those model objects in place, rather than replace those model objects with brand-new instances. The latter approach breaks data binding in general, but it is a much bigger problem when trying to update your UI from those models.

Observable Primitives

The entire model object itself does not have to be `Observable`. Whatever your binding expressions use, in terms of data, has to be `Observable`. That could be individual fields, if you are willing to publish those fields as `Observable` objects, such as by having them be `public final`.

An easy way to make a field be `Observable`, if the field is a primitive value (e.g., `int`), is to replace the field with the equivalent `Observable...` class (e.g., `ObservableInt`):

```
public final ObservableInt score=new ObservableInt();
```

Your code can use `get()` and `set()` methods on the `Observable...` primitive wrappers to get and set the primitive value itself. Calling `set()` also notifies all registered observers that the data has changed, and the data binding system uses that to find out that it needs to update your UI.

While this may sound a bit clunky, Java developers have used this pattern in other places. A common example are the `Atomic...` classes (e.g., `AtomicInteger`), that make modifying a primitive be guaranteed to be atomic, when that value might be get and set on multiple parallel threads.

ObservableField

For non-primitive values, but where the entire value changes in unison, you can use the generic `ObservableField` approach. In particular, a `String` is not a primitive, yet it is immutable, so changing the value means replacing the old `String` object with a new `String` object. `ObservableField` lets you set up observable strings:

```
  public final ObservableField<String> title=
    new ObservableField<String>();
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/Question.java)

This only works when you are replacing the entire object with a new object. So, for example, wrapping a `Location` in an `ObservableField` only works if you change the location by replacing the `Location`, instead of calling `setLatitude()` and `setLongitude()` on the existing `Location`. Replacing the `Location` outright triggers `ObservableField` to tell observers about the change. In contrast, `ObservableField` has no way to know that you called a method on the wrapped object that changes its state in a way that observers need to know about.

ObservableArrayList and ObservableArrayMap

The data binding system ships with two `Observable` classes that are collections.

One, `ObservableArrayList`, is fairly straightforward: it lets you add and remove members of the list, and it informs observers about those changes. Once again, it has no means of knowing if you change the state of a given list member, only if you change the state of the list itself.

The other is `ObservableArrayMap`. Android added the `ArrayMap` class in API Level 19. Functionally, `ArrayMap` works like a `HashMap`, as a collection of values accessed via keys, albeit with some additional APIs for working with the contents by numerical index, as you see with `ArrayList`. The implementation, though, trades off CPU time for memory efficiency. `ObservableArrayMap` adds `Observable` characteristics, such that changes to the contents of the `ArrayMap` are reported to observers.

Custom Observables

You can create your own class implementing the `Observable` interface. Most likely, you would do that by extending `BaseObservable`.

On the one hand, this does not have to be too complicated. For example, here is the implementation of `ObservableBoolean` from the data binding support library:

```
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
```

```
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package android.databinding;
import android.os.Parcel;
import android.os.Parcelable;
import java.io.Serializable;
/**
 * An observable class that holds a primitive boolean.
 * <p>
 * Observable field classes may be used instead of creating an Observable object:
 * <pre><code>public class MyDataObject {
 *     public final ObservableBoolean isAdult = new ObservableBoolean();
 * }</code></pre>
 * Fields of this type should be declared final because bindings only detect changes in the
 * field's value, not of the field itself.
 * <p>
 * This class is parcelable and serializable but callbacks are ignored when the object is
 * parcelled / serialized. Unless you add custom callbacks, this will not be an issue because
 * data binding framework always re-registers callbacks when the view is bound.
 */
public class ObservableBoolean extends BaseObservable implements Parcelable, Serializable {
    static final long serialVersionUID = 1L;
    private boolean mValue;
    /**
     * Creates an ObservableBoolean with the given initial value.
     *
     * @param value the initial value for the ObservableBoolean
     */
    public ObservableBoolean(boolean value) {
        mValue = value;
    }
    /**
     * Creates an ObservableBoolean with the initial value of <code>false</code>.
     */
    public ObservableBoolean() {
    }
    /**
     * @return the stored value.
     */
    public boolean get() {
        return mValue;
    }
    /**
     * Set the stored value.
     */
    public void set(boolean value) {
        if (value != mValue) {
            mValue = value;
            notifyChange();
        }
    }
    @Override
    public int describeContents() {
        return 0;
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(mValue ? 1 : 0);
```

```
    }
    public static final Parcelable.Creator<ObservableBoolean> CREATOR
            = new Parcelable.Creator<ObservableBoolean>() {
        @Override
        public ObservableBoolean createFromParcel(Parcel source) {
            return new ObservableBoolean(source.readInt() == 1);
        }
        @Override
        public ObservableBoolean[] newArray(int size) {
            return new ObservableBoolean[size];
        }
    };
}
```

A lot of that code is dealing with making `ObservableBoolean` be `Parcelable`. The key, from the standpoint of `BaseObservable`, is the call to `notifyChange()` in the `set()` method. This tells `BaseObservable` to tell all observers that stuff inside this `Observable` changed, and if they are tied to this `Observable`, they should go do something. Usually, "do something" will be to re-evaluate a binding expression and update a property of a `View`, such as updating the text of a `TextView` where a binding expression was used in the `android:text` attribute.

However, creating more complex custom observables is not especially well documented, and so we will explore that more later in this chapter.

An Observable Example

With all that behind us, let's look at another rendition of the Stack Overflow sample.

There are lots of values that are published for questions via the Stack Exchange API, beyond the ones used so far. One is the score, representing the net of upvotes and downvotes on the question. Of the question properties that we had been using before, only the title has a chance of changing in real time, and that does not happen very often. On the other hand, scores are far more likely to change on the fly.

So, the `DataBinding/Scored` sample project starts from the `DataBinding/Static` project and adds in support for the score property. It also makes the title and score `Observable` and adds a refresh action bar item. Tapping that item will update the data for the questions loaded in the app; any changes to titles or scores will be reflected directly, without additional code, by updating the models.

Of course, this sample app was not written with data binding in mind. While the previous two samples added on bits of data binding without significantly changing the app, this time we will have to take a chainsaw to our code to get what we want.

The Limitations of Earlier Examples

The specific problem we have to work around is the nature of our data model.

The previous versions of this sample would request the model objects via Retrofit and then slap them into an adapter to show in the `ListView`. From that point onward, the models were static — no code existed to add new questions, modify existing questions, etc.

However, Retrofit is designed to create new model objects on every call to a Web service interface. So, if we call once to get the latest questions, and then make another call to get updated versions of those questions, we wind up with two separate collections of model objects.

If we were not trying to use data binding, we could take a "caveman" approach: just replace the contents of the adapter with the new model collection. This would work, albeit with some impacts on the user experience (e.g., perhaps scrolling the list back to the top).

However, with data binding, we are effectively tying our *original* data model objects to our views more tightly. This means that when we get a new set of model objects from Retrofit, we cannot use them directly. Instead, we have to use them as a source of data, to be poured into our original model objects. Through the `Observable` mechanism, we can update the original models and not worry about the `ListView` rows, as data binding will take care of that for us. But this does mean that we need to have one "magic" set of model objects that represent the bound data, distinct from any model objects representing updates to that data.

Questions vs. Items

We could address the above problem by giving `Item` the ability to update its state from another `Item`. Our original query to get the most recent questions would create a collection of `Item` objects that would be our "durable" model, the one that we bind our UI to. Later updates that create new `Item` objects would be used solely to update the original durable `Item` objects' contents, not replace those objects.

But now we run into another problem: the `Observable` requirements of the data binding system may run counter to requirements imposed elsewhere.

In the case of this sample, `Item` is being populated by Gson, after Retrofit receives the JSON response from the server. Gson does not know anything about

`ObservableField`, `ObservableInt`, or any such things. There are two main approaches for dealing with this problem:

1. Use Gson's system of type adapters to try to teach Gson how to take JSON properties and update corresponding `ObservableField`, `ObservableInt`, etc. fields in the model. Most likely, this is the right direction for long-term use, though it is conceivable that something about Gson has irreconcilable differences with something about observable elements.
2. Have separate "model" objects. One represents the result of the Web service call (and gets populated by Gson), while the other represents the durable model (and has observable properties).

This revised edition of the sample takes the second approach. There is a new model class, `Question`, which models a Stack Overflow question. Our data binding will be applied to `Question`. `Item` is still there, but it represents the response from the Stack Exchange Web service call.

Keeping Score (and the ID)

Beyond dealing with the duality of `Question` and `Item`, we have two more JSON properties from the Web service response that we need to track. One is the `score`, as mentioned earlier. The other is the `question_id`, a unique ID for the question. We need this in order to be able to update an existing `Question` with data from a new `Item`, when we retrieve updates for our models.

The easy part is getting the new data from Retrofit and Gson. We just need to add two more fields to `Item`, for the score and question ID:

```
package com.commonsware.android.databind.basic;

import com.google.gson.annotations.SerializedName;

public class Item {
  String title;
  Owner owner;
  String link;
  int score;
  @SerializedName("question_id") String id;
}
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/Item.java)

In the case of the question ID, the JSON property is `question_id`. In Java, we will use `id` instead, using Gson's `@SerializedName` annotation to teach Gson to fill `question_id` properties into the `id` field.

We now also have a `Question` class that will be our observable, durable data model:

```
package com.commonsware.android.databind.basic;

import android.databinding.ObservableField;
import android.databinding.ObservableInt;

public class Question {
  public final ObservableField<String> title=
    new ObservableField<String>();
  public final Owner owner;
  public final String link;
  public final ObservableInt score=new ObservableInt();
  public final String id;

  Question(Item item) {
    updateFromItem(item);
    owner=item.owner;
    link=item.link;
    id=item.id;
  }

  void updateFromItem(Item item) {
    title.set(item.title);
    score.set(item.score);
  }
}
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/Question.java)

It holds the same five values as does `Item`, except that `title` and `score` are now `Observable`, via `ObservableField` and `ObservableInt`, respectively. The `owner`, `link`, and `id` values should be immutable, and we are not binding on them anyway, so keeping them as ordinary fields is fine.

`Question` has a constructor and an `updateFromItem()` method that both copy data from a `Item` into the `Question`. `updateFromItem()` handles the two `Observable` fields, and we will use this when we eventually fetch updates to the question. The constructor calls `updateFromItem()` plus populates the three `final` non-observable fields.

`QuestionsFragment` now has a more apropos name, as we will have it show the list of `Question` objects. Among other things, this requires changes to `QuestionsAdapter`, to work off of `Question` objects instead of `Item` objects:

```
class QuestionsAdapter extends ArrayAdapter<Question> {
  QuestionsAdapter(List<Question> items) {
    super(getActivity(), R.layout.row, R.id.title, items);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    RowBinding rowBinding=
      DataBindingUtil.getBinding(convertView);

    if (rowBinding==null) {
      rowBinding=
        RowBinding.inflate(getActivity().getLayoutInflater(),
          parent, false);
    }

    Question question=getItem(position);
    ImageView icon=rowBinding.icon;

    rowBinding.setQuestion(question);

    Picasso.with(getActivity()).load(question.owner.profileImage)
           .fit().centerCrop()
           .placeholder(R.drawable.owner_placeholder)
           .error(R.drawable.owner_error).into(icon);

    return(rowBinding.getRoot());
  }
}
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

Similarly, the `<variable>` in `row.xml` needs to be a `Question` now:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">

  <data>

    <import type="android.text.Html"/>

    <variable
      name="question"
```

```
      type="com.commonsware.android.databind.basic.Question"/>
  </data>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="@dimen/icon"
      android:layout_height="@dimen/icon"
      android:layout_gravity="center_vertical"
      android:contentDescription="@string/icon"
      android:padding="8dip"/>

    <TextView
      android:id="@+id/title"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical"
      android:layout_weight="1"
      android:text="@{Html.fromHtml(question.title)}"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/score"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:layout_marginLeft="8dp"
      android:layout_marginRight="8dp"
      android:text="@{Integer.toString(question.score)}"
      android:textSize="40sp"
      android:textStyle="bold"/>

  </LinearLayout>
</layout>
```

(from DataBinding/Scored/app/src/main/res/layout/row.xml)

You will note that the binding expression for the `score` `TextView` is `@{Integer.toString(question.score)}`. That is because the `score` field on `Question` is an `int`, and by default, the data binding system will think that is a reference to a string resource. We have to convert the score into a `String` to get the results that we want. We will see this more later in this chapter.

Refreshing the Data

Of course, having a `QuestionsAdapter` that adapts `Question` object only works if we *have* `Question` objects.

`QuestionsFragment` now holds onto two collections of `Question` objects: an `ArrayList` in the order that we get them from the Web service API, and a `HashMap` to find a `Question` object given its ID:

```
private ArrayList<Question> questions
  =new ArrayList<Question>();
private HashMap<String, Question> questionMap=
  new HashMap<String, Question>();
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

We could use a single `ArrayMap` or `SimpleArrayMap` instead, as that structure supports both indexed and keyed access. However, we would then have to roll our own `BaseAdapter`, as `ArrayAdapter` does not know how to work with `ArrayMap` or `SimpleArrayMap`. In this case, with just one group of 25 questions, having the less-efficient approach of two collections is simpler.

Our call to the `questions()` method on our `StackOverflowInterface` still returns a collection of `Item` objects. In `onCreateView()`, where we call `questions()`, we arrange to use those `Item` objects to create the corresponding group of `Question` objects:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=
      super.onCreateView(inflater, container,
        savedInstanceState);

  so.questions("android").enqueue(new Callback<SOQuestions>() {
    @Override
    public void onResponse(Call<SOQuestions> call,
                           Response<SOQuestions> response) {
      for (Item item : response.body().items) {
        Question question=new Question(item);

        questions.add(question);
        questionMap.put(question.id, question);
      }
```

```
      setListAdapter(new QuestionsAdapter(questions));
    }

    @Override
    public void onFailure(Call<SOQuestions> call, Throwable t) {
      onError(t);
    }
  });

  return(result);
}
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

That is sufficient to get our app to run again, showing the scores along with the question titles and asker avatars:

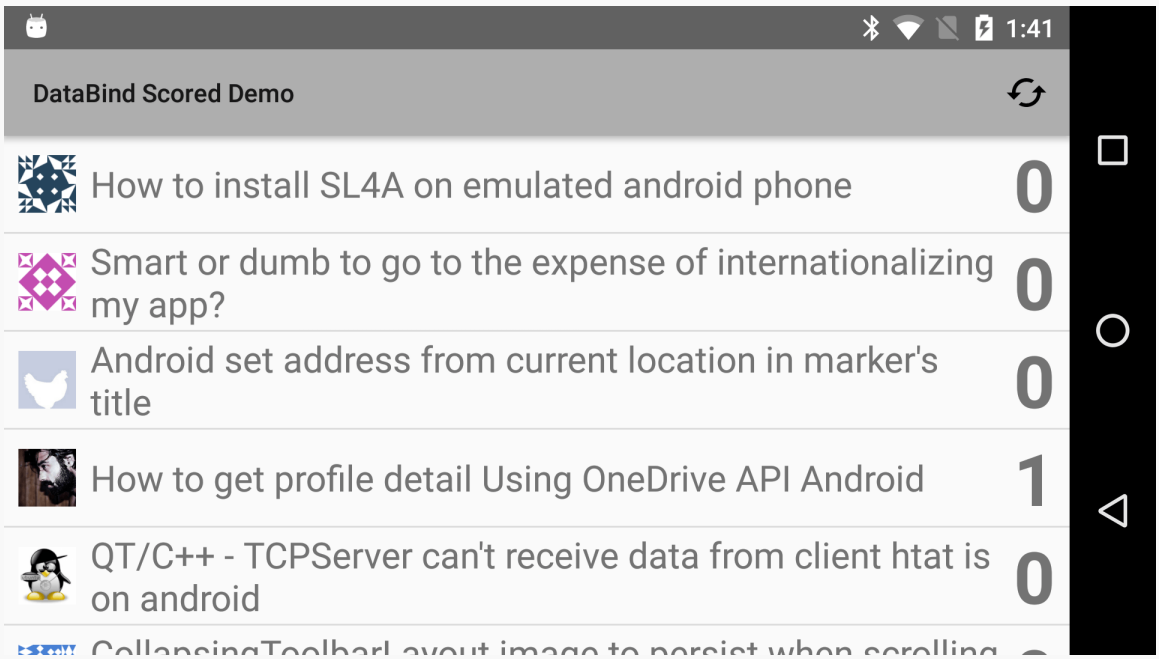


Figure 706: Stack Overflow Questions with Scores

However, we wanted to allow the user to refresh the data for these questions, so we can see a `score` being updated in real time via the data binding system. That requires a different call to the Stack Exchange API. It is still `/2.1/questions`, but now we have an additional path segment, one that takes a semi-colon-delimited list of question IDs. So, we add a new `@GET` method to `StackOverflowInterface` for this:

```
package com.commonsware.android.databind.basic;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Path;
import retrofit2.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  Call<SOQuestions> questions(@Query("tagged") String tags);

  @GET("/2.1/questions/{ids}?site=stackoverflow")
  Call<SOQuestions> update(@Path("ids") String tags);
}
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/StackOverflowInterface.java)

Note the use of `@Path("ids")` on the first parameter, corresponding to the `{ids}` placeholder in the path expressed in the `@GET` annotation. `@Path("ids")` says "the following parameter can be injected as a path segment into the URL", and `{ids}` indicates specifically where that parameter's value should go. Note, though, that it is a `String`, not a `String` array or `ArrayList` of strings. That is because we do not have a way to teach Retrofit how to concatenate a collection of strings into a single path segment.

In addition, this sample now has a `menu` resource directory, with an `actions.xml` resource in it, defining a single "refresh" menu item. The `QuestionsFragment` opts into participating in the action bar and, in `onCreateOptionsMenu()`, applies the `actions` menu resource. In `onOptionsItemSelected()`, if the user chose our refresh menu item, we call a private `updateQuestions()` method. This method needs to use the new `update()` method on `StackOverflowInterface` to update our collection of questions:

```
  private void updateQuestions() {
    ArrayList<String> idList=new ArrayList<String>();

    for (Question question : questions) {
      idList.add(question.id);
    }

    String ids=TextUtils.join(";", idList);

    so.update(ids).enqueue(new Callback<SOQuestions>() {
      @Override
      public void onResponse(Call<SOQuestions> call,
```

```
                          Response<SOQuestions> response) {
      for (Item item : response.body().items) {
        Question question=questionMap.get(item.id);

        if (question!=null) {
          question.updateFromItem(item);
        }
      }
    }

    @Override
    public void onFailure(Call<SOQuestions> call, Throwable t) {
      onError(t);
    }
  });
}
```

(from DataBinding/Scored/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

We collect all of the question IDs, then use `TextUtils.join()` to give us a single `String` with all the question IDs concatenated with semicolons. That, in turn, is passed to `update()`. For each returned `Item`, we find the corresponding `Question` in the `HashMap` and update it with the new data from the `Item`.

What we do *not* do is touch our UI.

However, if you run the app, choose a good question out of the list of questions, upvote the question, and refresh the list, you will see the new `score` appear immediately after the refresh. The data binding system handled that for us, without additional manual intervention on our part.

Two-Way Binding

So far, the focus has been on getting data from models into views. That is the most common scenario, as usually a subset of views accept user input, and plenty of user interfaces are read-only.

Plus, the original version of the data binding system only handled populating views from models.

But, in 2016, the data binding system was updated with "two-way binding", where views can populate models, in addition to having models populate views. While this

feature is presently undocumented, we have some limited information on how to make it work.

The change to the layout resources is very simple: use `@=` instead of `@`: `android:checked="@={question.expanded}"`.

This configures the attribute (the checked state of a `CompoundButton`) with the initial value of the `expanded` property on a `question` variable. It also *updates* the property if the user checks or unchecks the `CompoundButton`.

To make this work, you cannot use a simple `public` field for the property. It needs to either have a setter method (e.g., `setExpanded()`) or be a `public` `Observable` field.

For example, the `DataBinding/TwoWay` sample project is a clone of the `DataBinding/Scored` sample project from earlier in this chapter. However, now the `Question` will track some local state, information not obtained from the Stack Exchange API. Specifically, it will track a `boolean` value named `expanded`:

```
package com.commonsware.android.databind.basic;

import android.databinding.ObservableBoolean;
import android.databinding.ObservableField;
import android.databinding.ObservableInt;

public class Question {
  public final ObservableField<String> title=
    new ObservableField<String>();
  public final Owner owner;
  public final String link;
  public final ObservableInt score=new ObservableInt();
  public final String id;
  public ObservableBoolean expanded=new ObservableBoolean(true);

  Question(Item item) {
    updateFromItem(item);
    owner=item.owner;
    link=item.link;
    id=item.id;
  }

  void updateFromItem(Item item) {
    title.set(item.title);
    score.set(item.score);
  }
}
```

(from DataBinding/TwoWay/app/src/main/java/com/commonsware/android/databind/basic/Question.java)

Our `row` layout resource now has a `Switch` widget, bound to the `expanded` property using the `@=` syntax shown above:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <data>

    <import type="android.text.Html" />

    <variable
      name="question"
      type="com.commonsware.android.databind.basic.Question" />

    <variable
      name="controller"
      type="com.commonsware.android.databind.basic.QuestionController" />
  </data>

  <android.support.v7.widget.CardView xmlns:cardview="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    cardview:cardCornerRadius="4dp">

    <LinearLayout
      android:id="@+id/row_content"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="?android:attr/selectableItemBackground"
      android:gravity="center_vertical"
      android:onClick="@{()->controller.showQuestion(question)}"
      android:onTouch="@{(v,event)->controller.onTouch(v,event)}"
      android:orientation="horizontal">

      <Switch
        android:id="@+id/expanded"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="@={question.expanded}" />

      <ImageView
        android:id="@+id/icon"
        android:layout_width="@dimen/icon"
        android:layout_height="@dimen/icon"
        android:layout_gravity="center_vertical"
        android:contentDescription="@string/icon"
        android:padding="8dip"
        app:error="@{@drawable/owner_error}"
        app:imageUrl="@{question.owner.profileImage}"
        app:placeholder="@{@drawable/owner_placeholder}" />

      <TextView
        android:id="@+id/title"
        android:layout_width="0dp"
```

```
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"
        android:layout_weight="1"
        android:text="@{Html.fromHtml(question.title)}"
        android:textSize="20sp" />

      <TextView
        android:id="@+id/score"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:text="@{Integer.toString(question.score)}"
        android:textSize="40sp"
        android:textStyle="bold" />

    </LinearLayout>
  </android.support.v7.widget.CardView>
</layout>
```

(from DataBinding/TwoWay/app/src/main/res/layout/row.xml)

If you run the sample project, all of the switches will be checked at the outset, as we are defaulting `expanded` to `true`. If you uncheck some of them, and scroll around, you will see that the checked/unchecked state is handled properly, even though rows are being recycled along the way. And we did not have to add any Java code, other than the new property — in particular, neither our `ViewHolder` nor our `Adapter` need to worry about the `Switch`.

Other Features of Note

There are a number of other "bells and whistles" that you can utilize in the data binding system.

Obtaining Views via the Binding Class

The sample apps have been retrieving the `ImageView` widget for the row from the `RowBinding`. Any `View` in the layout file that has an `android:id` value will have a corresponding field in the `...Binding` generated class. So, for cases like the Picasso scenario, where we cannot use data binding to populate the `ImageView` and have to resort to classic bind-it-in-the-adapter logic, we do not have to do the `findViewById()` call ourselves. Instead, we just access the field in the binding class.

Manipulating Variables in the Binding

We have seen using a setter method to bind an object to a layout via the generated binding class. In the sample apps, we have been calling `setItem()` or `setQuestion()` to provide the model object to use in binding expressions. If needed, though, there is also a corresponding getter method (`getItem()`, `getQuestion()`) to retrieve the last-set value.

Views, Setters, and Binding

We have seen the use of `android:text` with a binding expression, to set the text for a `TextView`.

What really is going on is:

- The binding system evaluates the expression. This not only gives us the value to be bound, but also determines the data type of that value (e.g., `String`, `int`).
- The data binding system looks for a setter named `set...()`, where the `...` part is based on the name of the attribute (minus any namespace), where the data type matches the data type of the expression result. So, in the case where the binding expression generates a `String` for an `android:text` attribute, the data binding system will look for `setText(String)` on the widget, in our case a `TextView`. If the binding expression were to return an `int`, instead, the data binding system would look for `setText(int)`. In the case of `TextView`, that exists, and it is expecting the `int` to be a string resource. That is why, in the `Scored` sample app, we needed to convert the `int` to a `String`.

Of course, this is just the simple scenario.

Synthetic Properties

The data binding system maps attribute names to setters. But, what happens if you use an attribute name that does not actually exist?

Like the honey badger, the data binding system don't care.

All the data binding system is doing is using the attribute name to try to find an associated setter method. The fact that the attribute name is not actually part of the `LayoutInflater`-supported XML structure is irrelevant.

This means that you can use *any* attribute that maps to a setter method.

For example, `ViewPager` has no XML attributes of its own, beyond those it inherits from `View` or `ViewGroup`. But, you are welcome to use attributes like `app:currentItem` or `app:pageMargin` in your data binding-enhanced layout resources (where `app` points to a custom namespace of yours). `LayoutInflater` will parse them, but `ViewPager` will ignore them. However, the data binding system will happily let you bind values to them, triggering calls to `setCurrentItem()` and `setPageMargin()`, respectively.

Hence, do not feel that you are limited to only those attributes that are officially supported by `LayoutInflater` and the widgets. If the data binding system can find a setter, you can use it.

However, there is one key limitation with these synthetic properties: the value has to be a binding expression. That is true even if you are not really evaluating much of an expression.

For example, this will not work:

```
<ImageView
  android:id="@+id/icon"
  android:layout_width="@dimen/icon"
  android:layout_height="@dimen/icon"
  android:layout_gravity="center_vertical"
  android:contentDescription="@string/icon"
  android:padding="8dip"
  app:error="@drawable/owner_error"
  app:imageUrl="@{question.owner.profileImage}"
  app:placeholder="@drawable/owner_placeholder"/>
```

Here, we have three synthetic properties, `app:error`, `app:imageUrl`, and `app:placeholder`. Only `app:imageUrl` is using a binding expression, and its use of one makes sense, as we are pulling in data from a variable (`question`). The other two refer to drawables. Ideally, this would work. In practice, it does not work, as the binding system ignores the properties, and then Android complains that the attribute is not recognized.

This, however, works:

```
<ImageView
  android:id="@+id/icon"
  android:layout_width="@dimen/icon"
  android:layout_height="@dimen/icon"
  android:layout_gravity="center_vertical"
  android:contentDescription="@string/icon"
  android:padding="8dip"
  app:error="@{@drawable/owner_error}"
  app:imageUrl="@{question.owner.profileImage}"
  app:placeholder="@{@drawable/owner_placeholder}"/>
```

Now, `app:error` and `app:placeholder` use binding expressions... that happen to just return a drawable resource reference. This works, if one of two things are true:

1. There are setter methods for those properties (e.g., `setError()`) on `ImageView`, which in this case, there isn't, or
2. We use other techniques to tell the data binding system that those attributes get routed elsewhere, as will be seen in the next two sections

Using Different Methods

Of course, finding a setter may be a challenge. Frequently, the attribute name and the setter name follow the described convention (`android:foo` maps to `setFoo()`). Every now and then, though, the attribute name and setter name differ.

For example, `View` has an `android:fadeScrollbars` attribute, used to determine whether or not the scrollbars for a scrollable widget should automatically fade out after a stable period when the widget is not scrolling. However, the associated setter method is not `setFadeScrollbars()`, but instead `setScrollbarFadingEnabled()`. By default, in theory, the data binding system will not find the appropriate setter for `android:fadeScrollbars`.

In practice, the documentation suggests that Google has already fixed up all of the standard attributes from Android framework classes. However, there may still be gaps, particularly in Android Support-supplied classes, let alone third-party widgets.

To overcome the mis-matched attribute/setter pair, you can teach the data binding system how to find the setter for the attribute. To do this, you are supposed to be able to define a class-level `@BindingMethods` annotation, containing one or more `@BindingMethod` annotations, which in turn map an attribute on a type to a setter method name:

```
@BindingMethods({
  @BindingMethod(type = "android.view.View",
                 attribute = "android:fadeScrollbars",
                 method = "setScrollbarFadingEnabled"),
})
```

BindingAdapters, and the Picasso Scenario

Sometimes, even that is insufficient. Perhaps the setter method takes additional parameters, even though in your case they could be simply hard-coded or pulled from elsewhere in the widget. Perhaps the "setter method" is not really setting a property, but arranging to do some work related to the property.

For example, so far, we have not been able to use data binding with the `ImageView`. While the URL to the image is *related* to the `android:src` attribute, `android:src` does not take a URL, and we want to use Picasso to retrieve the image asynchronously anyway. Hence, we have been stuck with configuring the `ImageView` "the old-fashioned way" in `getView()`, by retrieving the `ImageView` and then telling Picasso how to populate it.

However, the data binding system can handle this too, by defining a custom `@BindingAdapter`.

Let's take a look at the `DataBinding/Picasso` sample project. This starts with the `Scored` sample from before, but now uses the data binding system to update the `ImageView`.

The `ImageView` XML from a little bit ago appears in our revised `row.xml` layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <data>

    <import type="android.text.Html"/>

    <variable
      name="question"
      type="com.commonsware.android.databind.basic.Question"/>
  </data>
```

```
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="@dimen/icon"
      android:layout_height="@dimen/icon"
      android:layout_gravity="center_vertical"
      android:contentDescription="@string/icon"
      android:padding="8dip"
      app:error="@{@drawable/owner_error}"
      app:imageUrl="@{question.owner.profileImage}"
      app:placeholder="@{@drawable/owner_placeholder}"/>

    <TextView
      android:id="@+id/title"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_gravity="left|center_vertical"
      android:layout_weight="1"
      android:text="@{Html.fromHtml(question.title)}"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/score"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:layout_marginLeft="8dp"
      android:layout_marginRight="8dp"
      android:text="@{Integer.toString(question.score)}"
      android:textSize="40sp"
      android:textStyle="bold"/>

  </LinearLayout>
</layout>
```

(from DataBinding/Picasso/app/src/main/res/layout/row.xml)

Here, we have three synthetic properties: attributes that are not really part of `ImageView`, but that we are using with the help of the data binding system.

To make that work, the data binding system has to know what to do with those three values. `ImageView` lacks setters for those, and so in the absence of anything

else, the data binding system will trigger a compilation error, complaining that it does not know what to do with the values we have in the layout.

To make this work, we need a `static` method somewhere, with the `@BindingAdapter` annotation. In this case, we have it defined on `QuestionsFragment`:

```
@BindingAdapter({"app:imageUrl", "app:placeholder", "app:error"})
public static void bindImageView(ImageView iv,
                                 String url,
                                 Drawable placeholder,
                                 Drawable error) {
  Picasso.with(iv.getContext())
    .load(url)
    .fit()
    .centerCrop()
    .placeholder(placeholder)
    .error(error)
    .into(iv);
}
```

(from DataBinding/Picasso/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

The method name does not matter, so call it whatever will help remind you of its role. It needs to return `void`, and take as parameters:

- the `View` type that the synthetic properties will appear on (in this case, `ImageView`)
- the values of those properties, in the order that they appear in the list of strings in the `@BindingAdapter` annotation

In our case, `app:placeholder` and `app:error` are resolving to `Drawable` resources, while `app:imageUrl` is resolving to a `String`.

This declaration teaches the data binding framework to call this method any time it finds a `View` of the designated type (`ImageView`) with the list of synthetic properties, instead of trying to find setter methods for those properties. Since the `<ImageView>` element in our layout file meets those criteria, the `bindImageView()` method will be called.

In that method, it is our job to do whatever it is that we need to do to consume those synthetic property values and apply their results to the supplied `View`. In this case, we have the snippet of Picasso code formerly found in the `getView()` method. However, before, the values of the drawables (placeholder and error) were hard-

coded in Java. Now, they are in the layout XML file, which is a bit more flexible, particularly if we are using different layout resources for different configurations.

This means we can junk the last of the manual binding code from `getView()`, leaving behind only the connection from our `ArrayAdapter` to the `RowBinding`:

```
  class QuestionsAdapter extends ArrayAdapter<Question> {
    QuestionsAdapter(List<Question> items) {
      super(getActivity(), R.layout.row, R.id.title, items);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      RowBinding rowBinding=
        DataBindingUtil.getBinding(convertView);

      if (rowBinding==null) {
        rowBinding=
          RowBinding.inflate(getActivity().getLayoutInflater(),
            parent, false);
      }

      rowBinding.setQuestion(getItem(position));

      return(rowBinding.getRoot());
    }
  }
```

(from DataBinding/Picasso/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

Note, though, that to make this sample work, we needed to make one other change. `app:imageUrl` refers to the `profileImage` field on the `Owner` class. Formerly, that was package-private, which means that the data binding generated code could not access it. Instead, we had to make it `public`:

```
package com.commonsware.android.databind.basic;

import com.google.gson.annotations.SerializedName;

public class Owner {
  public @SerializedName("profile_image") String profileImage;
}
```

(from DataBinding/Picasso/app/src/main/java/com/commonsware/android/databind/basic/Owner.java)

As an additional feature, a binding adapter can receive not only the new values for the properties, but the old ones as well (i.e., what had been used for a previous binding). To make that work, you double up all of the parameters, other than the `View` itself. First come the parameters that will be the old values, then come the parameters that will be the new values. If we wanted to use that in the sample shown in this section, we would have needed seven total parameters:

```
@BindingAdapter({"app:imageUrl", "app:placeholder", "app:error"})
public static void bindImageView(ImageView iv,
                                 String oldUrl,
                                 Drawable oldPlaceholder,
                                 Drawable oldError,
                                 String newUrl,
                                 Drawable newPlaceholder,
                                 Drawable newError) {
  // do good stuff here
}
```

For another example, the chapter on advanced keyboard and mouse support demonstrates a `BindingAdapter` to add a `focusMode` option to layouts, for a more flexible alternative to the `<requestFocus/>` XML element for controlling the widget that gets the focus.

Two-Way Binding and InverseBindingAdapter

Two-way binding works well in cases where the way you store the data in the models lines up well with the getters and setters of the associated widget. In the two-way binding example presented earlier, a `boolean` field in the model works well with the checked property of a `CompoundButton` like a `Switch`, as `CompoundButton` has an `isChecked()` method returning a `boolean` and a `setChecked()` accepting a `boolean`.

A `BindingAdapter` allows you to create other mappings between data types and properties, but only for the classic model->view binding. To accomplish the same thing in the reverse direction, you wind up creating an `InverseBindingAdapter`. As the name suggests, this serves the same basic role as a `BindingAdapter`, but in the inverse direction, taking data from the widget and preparing it for the model using custom code. Here, the "preparing it for the model" means converting it into a suitable data type for a setter, `Observable` field, etc. for your model.

This is fairly unusual.

The example used in some places is "what if I want to tie a `float` to an `EditText`?". The `InverseBindingAdapter` would look something like this:

```
@InverseBindingAdapter(attribute = "android:text")
public static float getFloat(EditText et) {
  try {
    return(Float.parseFloat(et.getText().toString()));
  }
  catch (NumberFormatException e) {
    return(0.0f); // because, um, what else can we do?
  }
}
```

The problem is if the user types in something that is not a valid floating-point number, like `snicklefritz`. `parseFloat()` will fail with a `NumberFormatException`. You *should* let the user know that their data entry was invalid. However, two-way data binding does not support this, with a default value (e.g., `0.0f`) being handed to the model instead.

Event Handling

So far, we have focused on binding expressions returning data that populates widgets, specifically by configuring how that widget looks.

But what about configuring how that widget *behaves*?

Whether this is a good idea is up for debate. On the one hand, it reduces the amount of boilerplate Java code necessary to wire up widgets. On the other hand, some might worry about a blurring of the lines separating views from things like controllers or presenters.

A 2016 update to the data binding system made it easier to set up these sorts of connections, though at the present time, this feature is undocumented.

Thinking Back to android:onClick

In the beginning, there was `android:onClick`, and it was good.

You could add the `android:onClick` attribute to a view in your layout resource XML, with a value of a method name in the activity that used the layout. That method needed to be `public`, return `void`, and take a `View` as a parameter — the same basic method signature as `onClick()` of an `OnClickListener`. When the user clicked the

view, the method named in `android:onClick` would be called, without having to call `setOnClickListener()` in Java with an `OnClickListener` implementation.

Over time, `android:onClick` faded in utility, as other things, such as fragments, started being where we wanted the click events to go. `android:onClick` could *only* call a method on the hosting activity, not a method on an arbitrary other class. No other attributes were created for other event handlers (long-click, touch, etc.), suggesting that this was a one-off experiment that would fade into oblivion.

And it did fade... until 2016, when the data binding system brought back the concept.

Tying Events to Methods Directly

For most events that you will care about with views, you can use a data binding expression to identify a method, on one of your variables, that will be called when the event is raised. Because this ties back to your variables, the method can be on any object that you inject into the binding, not just the activity.

It does make the syntax a bit more verbose. Instead of `android:onClick="doSomething"`, it becomes `android:onClick="@{controller::doSomething}"`, where `controller` is some object that you want to respond to the event (e.g., an MVC-style controller, an MVP-style presenter).

The methods referenced this way must have the same basic signature as the corresponding listener methods, just implemented on a custom class and with a custom name. So, for example, `onLongClick()` of an `OnLongClickListener` needs to return a `boolean`, indicating whether the event is consumed. If you use `android:onLongClick` to route that event to some custom method, that method must also return a `boolean`. Overall:

- The method must be `public`
- The method must take the same parameters as does the corresponding method on the regular listener class for this event
- The method must have the same return type as does the corresponding method on the regular listener class for this event

Tying Events to Methods via Lambda Expressions

Those restrictions on the methods tied in via data binding expressions can be a pain. In particular, you have no way of passing additional information from bound variables into the method, since those would not be part of the standard event handling method parameters.

However, the data binding system has another option for tying in event handlers: Java 8-style lambda expressions. So, you can have `android:onClick="@{()->controller::doSomething(thing)}"`, where `thing` is some variable in your layout resource, or a view (based on its `android:id` value), or the magic name `context` to provide a `Context`. It could also involve expressions using any of those as part of calculations (e.g., concatenating two strings).

You can also blend in parameters that are normally available to the event, such as `android:onClick="@{(v)->controller::doSomething(v, thing)}"`.

However, the argument list in the lambda function (the left-hand set of parentheses) either needs to be:

- empty, or
- have one entry for every parameter to the event handling method, even if you do not want all of those objects

For example, the `onCheckedChanged()` method on `OnCheckedChangeListener` for a `CompoundButton` takes two parameters: the `View` whose state changed, and a `boolean` indicating the new state. You cannot have `android:onCheckedChanged="@{(state)->controller::heyNow(state, thing)}"` or `android:onCheckedChanged="@{(view)->controller::heyNow(view, thing)}"`.

Instead, if you want either of those, you need to declare both, then just ignore the one that you do not need, such as `android:onCheckedChanged="@{(v, state)->controller::heyNow(state, thing)}"`.

Also, the method that you call still has to be `public` and still has to return the proper return type based on the event (e.g., `void` for `onClick`, `boolean` for `onLongClick()`).

With that in mind, the `DataBinding/RecyclerView` sample project demonstrates how this can work, along with how to use the data binding system to populate a `RecyclerView` instead of an `AdapterView`.

Converting to a RecyclerView/CardView UI

First, independent of data binding, we need to migrate the app over to use `RecyclerView`. Along the way, we can also add in support for `CardView`, to make the individual elements of the vertically-scrolling list look like cards, complete with rounded corners, drop shadows, and the like.

To that end, we add `recyclerview-v7` and `cardview-v7` to our roster of dependencies in `build.gradle`:

```
dependencies {
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
    implementation 'com.android.support:recyclerview-v7:25.1.0'
    implementation 'com.android.support:cardview-v7:25.1.0'
    implementation 'com.android.support:support-v4:25.1.0'
```

(from DataBinding/RecyclerView/app/build.gradle)

Our previous samples had used `ListFragment`. We do not have a `RecyclerViewFragment` given to us by the `recyclerview-v7` library. But, we can have our own, copied from one of the `RecyclerView` sample projects:

```
package com.commonsware.android.databind.basic;

import android.app.Fragment;
import android.os.Bundle;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class RecyclerViewFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
    RecyclerView rv=new RecyclerView(getActivity());

    rv.setHasFixedSize(true);

    return(rv);
  }

  public void setAdapter(RecyclerView.Adapter adapter) {
    getRecyclerView().setAdapter(adapter);
```

```
  }

  public RecyclerView.Adapter getAdapter() {
    return(getRecyclerView().getAdapter());
  }

  public void setLayoutManager(RecyclerView.LayoutManager mgr) {
    getRecyclerView().setLayoutManager(mgr);
  }

  public RecyclerView getRecyclerView() {
    return((RecyclerView)getView());
  }
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/RecyclerViewFragment.java)

All this does is manage a `RecyclerView` on our behalf, including allowing us to manipulate the adapter and the layout manager.

The revised `QuestionsFragment` now inherits from that `RecyclerViewFragment`. We configure the `RecyclerView` in `onViewCreated()`, mostly just using the code from before, except that we also need to call `setLayoutManager()` to indicate how we want the items to be laid out — in this case, opting for a vertically-scrolling list:

```
  @Override
  public void onViewCreated(View view,
                            Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    setLayoutManager(new LinearLayoutManager(getActivity()));

    so.questions("android").enqueue(new Callback<SOQuestions>() {
      @Override
      public void onResponse(Call<SOQuestions> call,
                             Response<SOQuestions> response) {
        for (Item item : response.body().items) {
          Question question=new Question(item);

          questions.add(question);
          questionMap.put(question.id, question);
        }

        setAdapter(new QuestionsAdapter(questions));
      }

      @Override
```

```
    public void onFailure(Call<SOQuestions> call, Throwable t) {
      onError(t);
    }
  });
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

Our `QuestionsAdapter` also has to change, to be a `RecyclerView.Adapter`, instead of an `ArrayAdapter`:

```
class QuestionsAdapter
  extends RecyclerView.Adapter<QuestionController> {
  private final ArrayList<Question> questions;

  QuestionsAdapter(ArrayList<Question> questions) {
    this.questions=questions;
  }

  @Override
  public QuestionController onCreateViewHolder(ViewGroup parent,
                                               int viewType) {
    RowBinding rowBinding=
      RowBinding.inflate(getActivity().getLayoutInflater(),
        parent, false);

    return(new QuestionController(rowBinding));
  }

  @Override
  public void onBindViewHolder(QuestionController holder,
                               int position) {
    holder.bindModel(getItem(position));
  }

  @Override
  public int getItemCount() {
    return(questions.size());
  }

  Question getItem(int position) {
    return(questions.get(position));
  }
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

We take in the roster of questions in the constructor and stash that for later use. `getItemCount()` and `getItem()` simply access that roster of questions. Data binding takes places in `onCreateViewHolder()`, where we create the `RowBinding` and use that to set up a `QuestionController`. `QuestionController` is a subclass of `RecyclerView.ViewHolder` and serves as the local controller for the row in our list — we will look at `QuestionController` in greater detail shortly. `onBindViewHolder()` simply tells the `QuestionController` to bind to the supplied `Question` model object.

`RecyclerView.ViewHolder` requires the root `View` for the row be supplied to its constructor. So, in the `QuestionController` constructor, we call `getRoot()` to get that `View` from the `RowBinding` and supply that, along with stashing the `RowBinding` in a field:

```
private final RowBinding rowBinding;

public QuestionController(RowBinding rowBinding) {
  super(rowBinding.getRoot());

  this.rowBinding=rowBinding;
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/QuestionController.java)

And, in `bindModel()`, we use the `RowBinding` to bind our `Question`, so the binding expressions will pull the title, score, and so forth into our views:

```
void bindModel(Question question) {
  rowBinding.setQuestion(question);
  rowBinding.setController(this);
  rowBinding.executePendingBindings();
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/QuestionController.java)

In a 2016 Google I|O presentation on data binding, Google engineers recommend that if you use `RecyclerView`, as part of `onBindViewHolder()` processing, that you call `executePendingBindings()` on the binding (e.g., `RowBinding` in the case of this example). This forces the data binding framework to get all of the bindings set up immediately, rather than waiting until the natural time to do it.

In our case, we just tuck that call into the `bindModel()` method of `QuestionController`, shown above.

You will notice that we also call a `setController()` method on the `RowBinding`. This is in support of our event handling binding work, as you will see next.

What About the Event Listeners?

`QuestionController` has two event-related methods. One is `onTouch()`, for handling the ripple effect on Android 5.0+:

```
@Override
public boolean onTouch(View v, MotionEvent event) {
  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
    v
      .findViewById(R.id.row_content)
      .getBackground()
      .setHotspot(event.getX(), event.getY());
  }

  return(false);
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/QuestionController.java)

The other is `showQuestion()`, which, surprisingly enough, will be called when we want to show the actual question:

```
public void showQuestion(Question question) {
  EventBus.getDefault().post(new QuestionClickedEvent(question));
}
```

(from DataBinding/RecyclerView/app/src/main/java/com/commonsware/android/databind/basic/QuestionController.java)

It contains the `EventBus` logic to tell somebody to go show some specified `Question`.

Those are tied into our app via the data binding framework:

```
  <LinearLayout
    android:id="@+id/row_content"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/selectableItemBackground"
    android:onClick="@{()->controller.showQuestion(question)}"
    android:onTouch="@{controller::onTouch}"
    android:orientation="horizontal">
```

(from DataBinding/RecyclerView/app/src/main/res/layout/row.xml)

For `android:onTouch`, we use the method-reference approach, asking the data binding framework to call `onTouch()` on our `controller`. For `android:onClick`, we use the lambda expression approach, calling `showQuestion()` on our `controller`, passing in the `question` variable, so we have our `Question` to go show.

And that's it. No other changes are needed to tie in these events, either in the `QuestionController` or in the `QuestionsAdapter`.

Type Converters

The result of a binding expression gets cast to the data type expected by the setter, field, or binding adapter that the data binding system identified as being the one to use.

Hopefully, this works.

However, it is possible that you will need to change your binding expression, such as in the case cited earlier in this chapter, where `android:text` can accept an integer, but you wanted that integer to be shown as text, not be a reference to a string resource.

In other cases, there may not be a clear match. Google's documentation cites the case where your binding expression returns the ID of a color resource, but the setter takes a `Drawable`, such as is the case with `setBackground()` on `View`.

One way of addressing this disparity is via a `@BindingMethod`. This teaches the data binding system to use a different method for the setter (e.g., `setBackgroundColor()`). However, this is *always* used for that particular widget class and attribute combination. In the particular case of the `android:background` attribute, there are a variety of possible setters:

- `setBackground(Drawable)`
- `setBackgroundColor(int)` (taking the actual color, not a color resource)
- `setBackgroundDrawable(Drawable)` (as `setBackground(Drawable)` is new to API Level 16)
- `setBackgroundResource(int)`

You may not be in position to use one of these for `android:background` exclusively.

Hence, another approach is to teach the data binding system how to convert data from one type to another, using a `@BindingConversion`-annotated `static` method:

```
@BindingConversion
public static ColorDrawable colorToDrawable(int color) {
   return new ColorDrawable(color);
}
```

As with binding adapters, the name of the method does not matter. What matters is that it takes an `int` as input and returns a `ColorDrawable`. The data binding system will take this into account and use it if it has a case where the binding expression returned an `int` and it needs a `ColorDrawable`... or a `Drawable`.

Here, though, we start to run into problems with Google's insistence on using `int` values everywhere. This `colorToDrawable()` conversion method takes an `int`. That `int` could be a color. It could be a color resource ID, or a string resource ID, or a layout resource ID, or the score of a Stack Overflow question, or countless other things. The depicted `@BindingConversion`, therefore, may not be especially useful.

Another scenario for `@BindingConversion` is to be able to extract something from deep inside a model without exposing the whole model structure as `public`. For example, the DataBinding/Conversion sample project uses a `@BindingConversion` to allow an `Owner` to be turned into a `String`, by means of returning the `profileImage` value:

```
@BindingConversion
public static String ownerToString(Owner owner) {
  return(owner.profileImage);
}
```

(from DataBinding/Conversion/app/src/main/java/com/commonsware/android/databind/basic/QuestionsFragment.java)

Once again, the method name does not matter; what matters is that this conversion knows how to handle taking an `Owner` and returning a `String`.

Now, the `app:imageUrl` attribute in the `ImageView` in the layout can refer to `question.owner` instead of `question.owner.profileImage`:

```
    <ImageView
      android:id="@+id/icon"
      android:layout_width="@dimen/icon"
      android:layout_height="@dimen/icon"
      android:layout_gravity="center_vertical"
      android:contentDescription="@string/icon"
      android:padding="8dip"
      app:error="@{@drawable/owner_error}"
```

```
      app:imageUrl="@{question.owner}"
      app:placeholder="@{@drawable/owner_placeholder}"/>
```

(from DataBinding/Conversion/app/src/main/res/layout/row.xml)

Chained Expressions

The original edition of the data binding system allowed you to create expressions based on variables and static methods. An update to data binding in 2016 added in "chained expressions", where expressions can refer to attributes of other widgets in the same layout resource. While this feature is presently undocumented, the basics are straightforward enough: just refer to the widgets by ID.

For example, the `DataBinding/Chained` sample project is a clone of the `DataBinding/TwoWay` sample project from earlier in the chapter. There, we added a `Switch` widget tied to an `expanded` property on the `Question` model objects. The reason for the name "expanded" was in preparation for the `DataBinding/Chained` sample, where the visibility of the avatar icon and the score would be toggled based on the `Switch` status.

The `Switch` has an `android:id` of `expanded`:

```
    <Switch
      android:id="@+id/expanded"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:checked="@={question.expanded}" />
```

(from DataBinding/Chained/app/src/main/res/layout/row.xml)

The `android:visibility` of the `icon` `ImageView` now is set based on a data binding expression, checking the `checked` state of the `expanded` widget, using a ternary operator to convert that into appropriate `View` values:

```
    <ImageView
      android:id="@+id/icon"
      android:layout_width="@dimen/icon"
      android:layout_height="@dimen/icon"
      android:layout_gravity="center_vertical"
      android:contentDescription="@string/icon"
      android:padding="8dip"
      android:visibility="@{expanded.checked ? View.VISIBLE : View.GONE}"
      app:error="@{@drawable/owner_error}"
      app:imageUrl="@{question.owner.profileImage}"
      app:placeholder="@{@drawable/owner_placeholder}" />
```

(from DataBinding/Chained/app/src/main/res/layout/row.xml)

Note that this requires us to import `View`, to be able to reference `View.VISIBLE` and `View.GONE`:

```
    <import type="android.view.View" />
```

(from DataBinding/Chained/app/src/main/res/layout/row.xml)

The `score` `TextView` could use the exact same expression as was used for the `icon` `ImageView`. However, in this case, the `visibility` of `score` depends upon the `visibility` of `icon`:

```
      <TextView
        android:id="@+id/score"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:text="@{Integer.toString(question.score)}"
        android:textSize="40sp"
        android:textStyle="bold"
        android:visibility="@{icon.visibility}" />
```

(from DataBinding/Chained/app/src/main/res/layout/row.xml)

This way, if the rules for how we derive the visibility change, all we need to do is change `icon`, leaving `score` alone.

Now, as the user toggles the `Switch`, the visibility of the `icon` and the `score` toggles with it.

Custom Binding Class Names

As noted earlier in this chapter, the binding class name for a layout resource is determined automatically by default. The layout filename is converted into a "Pascal case" rendition, then has `Binding` appended (e.g., `res/layout/foo_bar.xml` becomes `FooBarBinding`). This class goes in the `.databinding` sub-package under the base Java package for your app, as defined in the `package` attribute in your `<manifest>`.

However, this may result in awkward Java class names. Or, perhaps you want to have the classes be generated in some other Java package, for some reason. You can use

the `class` attribute on the `<data>` element to control the actual Java class name used for the binding class.

This can come in one of three forms:

- `class="Foo"` will name the binding class `Foo` and will place it in the standard `.databinding` sub-package
- `class=".Foo"` will name the binding class `Foo`, but will place it in the base package for your app (as defined in the `package` attribute), instead of in the separate `.databinding` sub-package
- `class="this.is.fully.qualified.Foo"` will name the binding class `Foo` and place it in the designated Java package

Extended Include Syntax

Android has supported `<include>` as a tag in layout resources since Android 1.0. The tag takes a `layout` attribute, pointing to a layout resource. The contents of the pointed-to layout resource are inserted into the view hierarchy of the original resource. So, if we have:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <include layout="@layout/foo"/>

    <!-- other widgets go here -->

</LinearLayout>
```

... then whatever is in the `foo` layout resource will be added to the `LinearLayout`, ahead of any other widgets in that `LinearLayout`.

With the data binding system, you can pass variables from the outer layout to the included one, without having to somehow bind the variable yourself in the included layout from Java code:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:bind="http://schemas.android.com/apk/res-auto">
  <data>
      <variable name="foo" type="com.thingy.Foo"/>
  </data>
```

```
  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent">

      <include layout="@layout/foo" bind:bar="@{foo}"/>

      <!-- other widgets go here -->

  </LinearLayout>
</layout>
```

Here, if the `foo` layout resource has a variable named `bar`, it will be populated by evaluating the `@{foo}` binding expression, so the `foo` resource can refer to `bar` in its own binding expressions.

Custom Observables

What you want may not fit any of these patterns. In that case, you are going to have to roll your own `Observable` implementation. The simplest way to do that is to extend `BaseObservable`, which handles all of the observer registration logic for you.

There are two types of changes for which you can notify observers:

- Changes to properties, which can be handled by the individual property observers described above, such as `ObservableField`
- Changes to other intrinsic aspects of the model itself, that cannot be captured in a simple `Observable` wrapper on some property

For example, you might have a `Person` class that has `birthDate` field, of type `Date`, representing the date on which the person was born. If you wanted to use that date in a binding expression, you could have `birthDate` be `public`, or have a `getBirthDate()` that returned it. If you wanted a binding expression to be updated when the birth date changed (e.g., correcting a typo), you could have `birthDate` be an `ObservableField` wrapped around a `Date`.

However, suppose what you *really* want to use in the binding expression is the person's age. It is easy enough for `Person` to calculate that, based on the current date and `birthDate`. However, this would be awkward to publish via an `ObservableField`, since there should not be an `age` field — age is a derived value, not a stored value. Instead, you could say that your `getAge()` method publishes a simple `int`, and you

will handle notifying observers whenever the age changes, either due to a change in `birthDate`, or if the date changed and it is now the person's birthday.

Bindable Properties

On a `BaseObserverable`, you can annotate getter-style methods with `@Bindable`. This tells the data binding framework that those methods represent values that can be bound. Because `BaseObservable` implements `Observable`, the data binding framework can call `addOnPropertyChangedCallback()` to register an `OnPropertyChangedCallback` to find out when `@Bindable` properties are changed.

To make that work, `BaseObservable` supplies a `notifyPropertyChanged()` method. You can call this from the setter method or other place where you are changing the value of the property, to let `BaseObservable` know that the property changed. This, in turn, will let all `OnPropertyChangedCallback` instances know about the change, which will trigger the data binding framework to re-evaluate any binding expressions tied to that property.

Unfortunately, this is broken in the 1.5.1 build of Android Studio and the 1.5.0 edition of the Android Gradle Plugin.

For example, here is a revised version of the `Question` model class that has it use `BaseObservable` and `notifyPropertyChanged()`:

```
package com.commonsware.android.databind.basic;

import android.databinding.BaseObservable;
import android.databinding.Bindable;
import com.commonsware.android.databind.basic.BR;

public class Question extends BaseObservable {
  private String title;
  private final Owner owner;
  private final String link;
  private int score;
  private final String id;

  Question(Item item) {
    updateFromItem(item);
    owner=item.owner;
    link=item.link;
    id=item.id;
  }
```

```
  @Bindable
  public String getTitle() {
    return(title);
  }

  @Bindable
  public Owner getOwner() {
    return(owner);
  }

  @Bindable
  public String getLink() {
    return(link);
  }

  @Bindable
  public int getScore() {
    return(score);
  }

  @Bindable
  public String getId() {
    return(id);
  }

  void updateFromItem(Item item) {
    this.title=item.title;
    this.score=item.score;

    notifyPropertyChanged(BR.title);
    notifyPropertyChanged(BR.score);
  }
}
```

Here, `BR` is a generated class. According to the documentation:

> The `Bindable` annotation generates an entry in the `BR` class file during compilation. The `BR` class file will be generated in the module package.

Unfortunately, while this is all true, Android Studio does not recognize any of the generated fields, and so while you can import `BR`, `BR.title` and `BR.score` — the `int` values identifying those properties – are not recognized and result in compile errors.

Notifying About Intrinsic Changes

If the `BaseObservable` itself is what is used in the binding expression, or if you want to use bindable properties and need to work around the `BR` issue mentioned above, `BaseObservable` also offers `notifyChange()`, indicating that all binding expressions tied to the `BaseObservable` instance should be re-evaluated.

The DataBinding/Observable sample project is another variation of the sample project that we have been analyzing in this chapter. This one has `Question` extend `BaseObservable`. However, unlike the code snippet above, where we tried using `BR` and `notifyPropertyChanged()`, here we just settle for `notifyChange()`:

```
package com.commonsware.android.databind.basic;

import android.databinding.BaseObservable;
import android.databinding.Bindable;
import android.databinding.ObservableField;
import android.databinding.ObservableInt;
import com.commonsware.android.databind.basic.BR;

public class Question extends BaseObservable {
  private String title;
  private final Owner owner;
  private final String link;
  private int score;
  private final String id;

  Question(Item item) {
    updateFromItem(item);
    owner=item.owner;
    link=item.link;
    id=item.id;
  }

  @Bindable
  public String getTitle() {
    return(title);
  }

  @Bindable
  public Owner getOwner() {
    return(owner);
  }

  @Bindable
  public String getLink() {
```

```
    return(link);
  }

  @Bindable
  public int getScore() {
    return(score);
  }

  @Bindable
  public String getId() {
    return(id);
  }

  void updateFromItem(Item item) {
    this.title=item.title;
    this.score=item.score;

    notifyChange();
  }
}
```

(from DataBinding/Observable/app/src/main/java/com/commonsware/android/databind/basic/Question.java)

Even though we are storing `title` as a simple `String` and `score` as a simple `int`, we can use them in binding expressions, because their getters are `@Bindable` and we are notifying `BaseObservable` when their values change.

Thinking Outside the Box

Data binding will usually be used for things like the text of a `TextView`, or the image shown in an `ImageView`. However, you are welcome to have other things vary based upon binding expressions. For example, perhaps you want a certain background color or color bar on a row in a list, based upon some category associated with the model objects. You could use data binding to set that color.

Lisa Wray pointed out another inventive use of data binding: custom fonts.

Historically, using a custom `Typeface` required Java code. That Java code might be fairly limited, if you only need to update one `TextView`. Or, that Java code might pull in a library like Calligraphy to be able to apply arbitrary fonts to arbitrary widgets from within layout files.

The data binding framework can handle that for you, if you create a custom `BindingAdapter` for some synthetic property (e.g., `wray:font`). In your layout, you

would have `wray:font` attributes that name the typeface that you want on relevant widgets (e.g., `TextView`):

```
<TextView
  wray:font="@{`MgOpenCosmetica.ttf`}"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"/>
```

The `BindingAdapter` would retrieve the `Typeface` for that font name, then apply it to the associated widget:

```
@BindingAdapter({"wray:font"})
public static void setFont(TextView tv, String font){
  String assetPath="fonts/" + font;
  Typeface type=Typeface.createFromAsset(tv.getContext().getAssets(), assetPath);

  tv.setTypeface(type);
}
```

This particular implementation has performance issues, as it creates a new `Typeface` object on every binding, which is inefficient. Lisa has a complete sample app that demonstrates caching the `Typeface` objects to reduce the performance overhead.

It is likely that the Android community will come up with other interesting tricks for simplifying code using fancy data binding adapters, converters, and the like.

Drag and Drop

Desktop applications have long offered drag-and-drop, both within and between applications. Android has supported this for quite some time, but you could only drag and drop within a single activity. As a result, this was not especially popular.

However, starting in Android 7.0, you can drag and drop between applications, so long as their windows are visible in a multi-window environment. Not only does this make drag-and-drop more compelling in general, but in a freeform multi-window environment, users will expect Android apps to behave like their desktop counterparts. Hence, users will expect drag-and-drop capabilities where it makes sense.

In this chapter, we will explore Android's drag-and-drop facility, including how to perform it between separate applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, as well as the chapter on the clipboard.

One example uses `RecyclerView`, so reviewing that chapter is a good idea. Similarly, one sample uses `StreamProvider`, so you may wish to read the section on it as well.

The Scope of Drag and Drop

Since the term "drag-and-drop" means different things to different people – including different developers used to different platforms — it will help if we understand exactly what Android's definition of "drag-and-drop" is.

What Are We Dragging and Dropping?

In Android, the focus is on dragging and dropping content, meaning some information identified by a `Uri` and an associated MIME type. We are using the drag-and-drop process to select some piece of content and inform something else about that content. Specifically, the content that we are dragging and dropping is represented by a `ClipData` object, the same as we could use with the clipboard.

Technically, the `ClipData` does not have to represent content. The clipboard supports plain text `ClipData` items, and nothing is stopping you from using drag and drop for plain text as a result. When dragging and dropping between apps, this may cause some compatibility issues, though the drag-and-drop framework takes steps to help deal with this. Within an app, options like plain text allow you to "cheat" to an extent, allowing drag-and-drop to support anything you want, so long as you can identify the specific "anything you want" by a string ID or key.

From the *user's* standpoint, the user is dragging some visual representation of this content. That can be whatever bitmap you want, and you will have a few options for specifying what this bitmap is. This bitmap is referred to as the "shadow".

Where Are We Dragging From?

You will need to provide some UI that triggers a drag-and-drop operation, not only allowing the user to say "let's drag this somewhere" but also "here is what 'this' I want to drag".

A typical trigger for this is a long-click. So, for example, a long-click on a list row might trigger a drag-and-drop of the content identified by that row.

Usually, the trigger is tied to some view, as drag-and-drop intrinsically is a visual operation. Technically, this is not required, if you can find some other approach that users will understand and appreciate.

Where Are We Dropping To?

You will need to identify possible drop targets, in the form of views. A view can be registered as a potential drop target, then stipulate whether it is a candidate for a *specific* drag-and-drop operation when that operation begins. For example, if you have two lists, and you want the user to drag items between the lists, both are potential drop targets. However, you might elect to say that the user cannot drag

from a list back into that same list, so if the content being dragged originated from the list, that list is not a candidate for that specific drag-and-drop operation.

The Pieces of Drag-and-Drop

As noted above, what we are really dragging and dropping is a `ClipData`, which can represent whatever we want, so long as the recipient of that `ClipData` knows how to work with whatever the provider of that `ClipData` put in it.

However, there are a few other pieces to the drag-and-drop process.

The Drag Shadow

The drag shadow is the visual representation of what the user is dragging and dropping. Programmatically, the shadow is defined as an instance of `View.DragShadowBuilder` (which, despite the name, does *not* implement a builder-style API).

You have two main choices for creating this shadow: use a `View`, or use a `Canvas`.

…From a View

You can create a `View.DragShadowBuilder` via the constructor that takes a `View` as a parameter. This tells `View.DragShadowBuilder` that the drag shadow should be a translucent copy of whatever the `View` is showing at the time we start the drag-and-drop operation.

This is very easy to implement, and it works well if you have a `View` that makes for a likely visual representation of what is being dragged and dropped.

On the other hand, it will not handle all scenarios. Suppose that you want to allow the user to drag from a list. Furthermore, suppose that you want the user to be able to multi-select items in the list and drag the entire selection. Now you no longer have a single `View` that you can use as the basis for the drag shadow.

Also, keep the drag shadow relatively small. It needs to be big enough that the user can see it despite a finger potentially being in the way. However, it also needs to be small enough to make it clear where the user is dropping it. This is another reason why the multi-select list scenario does not work well with creating a `View.DragShadowBuilder` from a `View` — even if you chose the `ListView` or

`RecyclerView` as being the `View` from which to create the drag shadow, odds are that the list will be far too large.

…From a Canvas

For cases where using a `View` as the basis of your drag shadow will not work, you can create your own subclass of `View.DragShadowBuilder` and define the drag shadow however you want.

To do this, you will override two methods. One is `onProvideShadowMetrics()`, where you fill in a pair of `Point` objects. The first represents the size of the drag shadow in pixels. The other represents the point within the drag shadow where the touch point will be — in other words, where is the drag shadow with respect to where the finger is touching the screen.

The other method is `onDragShadow()`, where you are given an appropriately-sized `Canvas` and you can draw whatever you want into that `Canvas` to serve as the drag shadow. For example, you might draw a `Bitmap` in `onDragShadow()` using the dimensions of the `Bitmap` and its center point in `onProvideShadowMetrics()`.

Technically, you can combine the two approaches. You create the `View.DragShadowBuilder` using a `View` but then override one or both of the aforementioned `View.DragShadowBuilder` methods to alter the default behavior a bit. For example, by default, the touch point will be the center of the `View`, but you might want the touch point to be offset towards one corner — you could handle this by overriding `onProvideShadowMetrics()`, chaining to the superclass, then updating the second `Point` object as you see fit.

The Drag Event Listener

To react to drag events in a drop target `View`, you can call `setOnDragListener()`, supplying an implementation of `View.OnDragListener`. This interface has a single method, `onDrag()`, that you will need to implement.

The sample apps in this chapter implement `View.OnDragListener` on the activity that has the drop targets. Typically, you will implement `View.OnDragListener` on whatever object in your UI handles events raised by the widgets (e.g., a controller or presenter).

The Drag Events

`onDrag()` of your `View.OnDragListener` is passed two objects: the `View` that you called `setOnDragListener()` on, and a `DragEvent` representing what is happening with respect to the drag-and-drop process.

The `DragEvent` contains an action `int` value, representing what the state change is in the drag-and-drop operation. Depending on the action, other aspects of the `DragEvent` may be available to you as well.

ACTION_DRAG_STARTED

When the user begins a drag-and-drop operation, and your window is visible (e.g., the user started the drag-and-drop within your own activity), you will receive a `DragEvent` whose action is `ACTION_DRAG_STARTED`.

Your primary job is to return `true` from `onDrag()` if you wish to be considered a drop target for this drag-and-drop operation. Prior to Android 7.0, you might always return `true`, since you are certain to be in control over both the drag and the drop. Starting with Android 7.0, you might conditionally return `true`, if the drag-and-drop operation looks like it might be one that you can handle. Unfortunately, your primary means of determining this is via `getClipDescription()` on the `DragEvent`, which gives you a `ClipDescription` describing the `ClipData` that is the content. This does not give you much to go on, as we will see in upcoming samples.

If you are a valid drop target, you might also consider adjusting the look and feel of this `View` to indicate to the user that this is a valid drop target. Android does not do anything on its own for this. You might tint the `View`, or add an outline, or something, to help clue the user in that dropping over your `View` might have a positive result.

If you return `true`, you will be notified about the progress of the drag-and-drop event through the other event actions listed below. If you return `false`, you are indicating that this drag-and-drop operation does not concern you, and you will not be given any further `DragEvents` for it.

ACTION_DRAG_ENTERED

You will receive a `DragEvent` with this action once it is possible for the user to drop in your `View`. This will come when the drag shadow enters the bounding box of the `View` (not necessarily only where pixels are drawn for the view).

If you are still interested in this drag-and-drop operation, you should:

- Return `true` from `onDrag()`, and
- Alter the widget's appearance yet again, to reflect the fact that if the user lifts her finger, the content will be dropped into this widget

If precise placement within the widget is important for the drag-and-drop operation (e.g., you wish to highlight some specific cell in a grid), you can call `getX()` and `getY()` on the `DragEvent` to try to determine where the drop point is. Unfortunately, it is not documented whether `getX()` and `getY()` are relative to your widget, the screen, or something else.

ACTION_DRAG_LOCATION

If you return `true` from the `ACTION_DRAG_ENTERED` `DragEvent`, you may receive additional `DragEvents` with `ACTION_DRAG_LOCATION` actions, indicating that the user has moved within the bounding box of your widget. If you are using `getX()` and `getY()` to deal with the highlighting, these values will have changed, and so you will want to update the highlighting to match.

ACTION_DRAG_EXITED

If you return `true` from the `ACTION_DRAG_ENTERED` `DragEvent`, you may receive a `DragEvent` for `ACTION_DRAG_EXITED`. This indicates that the user dragged the item outside of your widget without dropping it. Any state changes to your widget, such as a highlight, that you applied in `ACTION_DRAG_ENTERED` or `ACTION_DRAG_LOCATION` should be reverted. However, the drag-and-drop operation is still proceeding, so any highlight you use for that (e.g., in `ACTION_DRAG_STARTED`) should still be used.

ACTION_DROP

Of course, the fun action is `ACTION_DROP`, which means that the user dropped the content over this widget as the drop target. You can call `getClipData()` to get at the `ClipData` for this content, along with final `getX()` and `getY()` values.

If you return `true` in `onDrag()`, this indicates that you handled the drop request.

However, you may not be able to handle the drop request. For example, suppose you are looking to have a `Uri` pointing to a video be dropped into your app. All you can determine from the `ClipDescription`, in your `ACTION_DRAG_STARTED` processing, is that the `ClipData` has a `Uri`. So, you have to return `true` from `onDrag()` in your `ACTION_DRAG_STARTED` logic. But, then, in `ACTION_DROP` processing, when you get the real `Uri`, you find out that it has a different MIME type (e.g., `text/html`, instead of `video/*`). You will need to return `false` from `onDrag()` in your `ACTION_DROP` logic. Unfortunately, what happens from this point forward is undocumented.

ACTION_DRAG_ENDED

If you returned `true` from the `ACTION_DRAG_STARTED` `DragEvent`, you should receive a `DragEvent` when the drag-and-drop operation is over, with `ACTION_DRAG_ENDED` as the action. Any state change you made to your widget in `ACTION_DRAG_STARTED` should be reverted here. If it matters to you whether the user did a valid drop or not, call `getResult()` on the `DragEvent` to find out .

Drag-and-Drop, within an Activity

The classic drag-and-drop scenario, prior to Android 7.0, was to drag-and-drop between widgets in a single activity.

The `DragDrop/Simple` sample project demonstrates this scenario. It is based on the `RecyclerView/VideoList` sample app from the chapter on `RecyclerView`.

The Landscape Layout

On smaller screens, we just have the `RecyclerView` as before. However, on larger screens (e.g., 9" tablets in landscape), we put a `VideoView` and an `ImageView` alongside the `RecyclerView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">

  <android.support.v7.widget.RecyclerView android:id="@+id/video_list"
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1" />

  <LinearLayout
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:orientation="vertical">

    <FrameLayout
      android:id="@+id/video_frame"
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_marginBottom="4dp"
      android:layout_weight="1"
      android:padding="4dp">

      <VideoView
        android:id="@+id/player"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center" />
    </FrameLayout>

    <FrameLayout
      android:id="@+id/thumbnail_frame"
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="1"
      android:padding="4dp">

      <ImageView
        android:id="@+id/thumbnail_large"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerInside" />
    </FrameLayout>
  </LinearLayout>
</LinearLayout>
```

(from DragDrop/Simple/app/src/main/res/layout-w800dp/main.xml)

The idea is that the user will be able to drag from the `RecyclerView` into the other two widgets, which will play the video or show a larger rendition of the thumbnail, respectively.

The `VideoView` and the `ImageView` are each wrapped in a `FrameLayout`. Mostly, that is to give us a place to render a border around the widgets, indicating that they are drop targets. We have a pair of `<shape>` drawables for this. One is a red dashed line indicating a potential drop target:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">
  <stroke
    android:width="2dp"
    android:dashGap="8dp"
    android:dashWidth="8dp"
    android:color="#ff0000" >
  </stroke>
</shape>
```

(from DragDrop/Simple/app/src/main/res/drawable-nodpi/droppable.xml)

The other is a solid green line indicating a "live" drop target, used to indicate that dropping the content here should work:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">
  <stroke
    android:width="4dp"
    android:color="#2e7d32" >
  </stroke>
</shape>
```

(from DragDrop/Simple/app/src/main/res/drawable-nodpi/drop.xml)

Registering as Drop Targets

This version of the sample app avoids the `RecyclerViewActivity` used in the `RecyclerView/VideoList` sample app. Instead, `MainActivity` manages all of its widgets directly, including the `RecyclerView`.

In `onCreate()`, after inflating the layout, we attempt to retrieve the `VideoView` and `ImageView`. If we find them, we call `setOnDragListener()`, supplying our `MainActivity` instance itself as the `OnDragListener` implementation:

```
    player=(VideoView)findViewById(R.id.player);

    if (player!=null) {
```

```
    player.setOnDragListener(this);
  }

  thumbnailLarge=(ImageView)findViewById(R.id.thumbnail_large);

  if (thumbnailLarge!=null) {
    thumbnailLarge.setOnDragListener(this);
  }
```

(from DragDrop/Simple/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

We will examine the `onDrag()` method that `OnDragListener` requires shortly.

Starting to Drag

This app supports a long-click on a row in our `RecyclerView` to enter drag-and-drop mode, as we call `setOnLongClickListener()` on the row itself, in `RowController` (our `RecyclerView.ViewHolder` for our list rows):

```
RowController(View row) {
  super(row);

  title=(TextView)row.findViewById(android.R.id.text1);
  thumbnail=(ImageView)row.findViewById(R.id.thumbnail);

  row.setOnClickListener(this);
  row.setOnLongClickListener(this);
}
```

(from DragDrop/Simple/app/src/main/java/com/commonsware/android/dragdrop/RowController.java)

In `onLongClick()`, we:

- Create a `ClipData` based on the `Uri` obtained from `MediaStore` for the video, plus its caption (pulled from the `title` `TextView`)
- Create a drag shadow, using `View.DragShadowBuilder`, with the `thumbnail` `ImageView` as the basis
- Call `startDrag()` on the row itself, accessed via the `itemView` field on the `ViewHolder` base class

```
@Override
public boolean onLongClick(View v) {
  ClipData clip=ClipData.newRawUri(title.getText(), videoUri);
  View.DragShadowBuilder shadow=new View.DragShadowBuilder(thumbnail);
```

```
    itemView.startDrag(clip, shadow, Boolean.TRUE, 0);

    return(true);
  }
```

(from DragDrop/Simple/app/src/main/java/com/commonsware/android/dragdrop/RowController.java)

Besides the `ClipData` and `View.DragShadowBuilder`, `startDrag()` takes two other parameters:

- An arbitrary `Object` referred to as the "local state", which can provide additional information between the drag source and the drop target, but only when both are in the same window (usually meaning the same activity)
- A set of flags (here unused, so set to `0`)

For the local state, we are using `Boolean.TRUE`. That is a fairly arbitrary choice, but it is a good idea to pass a non-`null` value here, for reasons that we will get into later in this chapter.

Reacting to Drag Events

Our `onDrag()` method in `MainActivity` will handle all of the events related to our registered drop targets:

```
@Override
public boolean onDrag(View v, DragEvent event) {
  boolean result=true;

  switch (event.getAction()) {
    case DragEvent.ACTION_DRAG_STARTED:
      if (event.getLocalState()==null) {
        result=false;
      }
      else {
        applyDropHint(v, R.drawable.droppable);
      }
      break;

    case DragEvent.ACTION_DRAG_ENTERED:
      applyDropHint(v, R.drawable.drop);
      break;

    case DragEvent.ACTION_DRAG_EXITED:
      applyDropHint(v, R.drawable.droppable);
      break;
```

```
    case DragEvent.ACTION_DRAG_ENDED:
      applyDropHint(v, -1);
      break;

    case DragEvent.ACTION_DROP:
      ClipData.Item clip=event.getClipData().getItemAt(0);
      Uri videoUri=clip.getUri();

      if (v==player) {
        player.setVideoURI(videoUri);
        player.start();
      }
      else {
        Picasso.with(thumbnailLarge.getContext())
          .load(videoUri.toString())
          .fit().centerCrop()
          .placeholder(R.drawable.ic_media_video_poster)
          .into(thumbnailLarge);
      }

      break;
  }

  return(result);
}
```

(from DragDrop/Simple/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

For most of the actions, we apply (or remove) a drawable from the `FrameLayout` containers wrapping our `VideoView` and `ImageView` widgets, via an `applyDropHint()` utility method:

```
private void applyDropHint(View v, int drawableId) {
  View parent=(View)v.getParent();

  if (drawableId>-1) {
    parent.setBackgroundResource(drawableId);
  }
  else {
    parent.setBackground(null);
  }
}
```

(from DragDrop/Simple/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

Here, we use `-1` as the "ID" of a resource meaning to remove any previous background.

In `onDrag()`, we ignore `ACTION_DRAG_LOCATION` events, as we are not using `getX()` and `getY()` and so do not care if those values change.

However, we do handle two actions a bit differently:

- `ACTION_DRAG_STARTED` examines the local state and rejects any `DragEvent` where that state is `null`, for reasons that we will get into later in this chapter
- `ACTION_DROP` retrieves the `ClipData.Item` for the `ClipData` we set as the drag content, retrieves the `Uri` of the video from the `ClipData.Item`, then either plays the video or shows the thumbnail, depending on which widget the user dropped the content into

The Result

If you run the sample app on a large-enough device with a roster of videos, you will get the list of videos on one side, and the empty `VideoView` and `ImageView` on the other side:

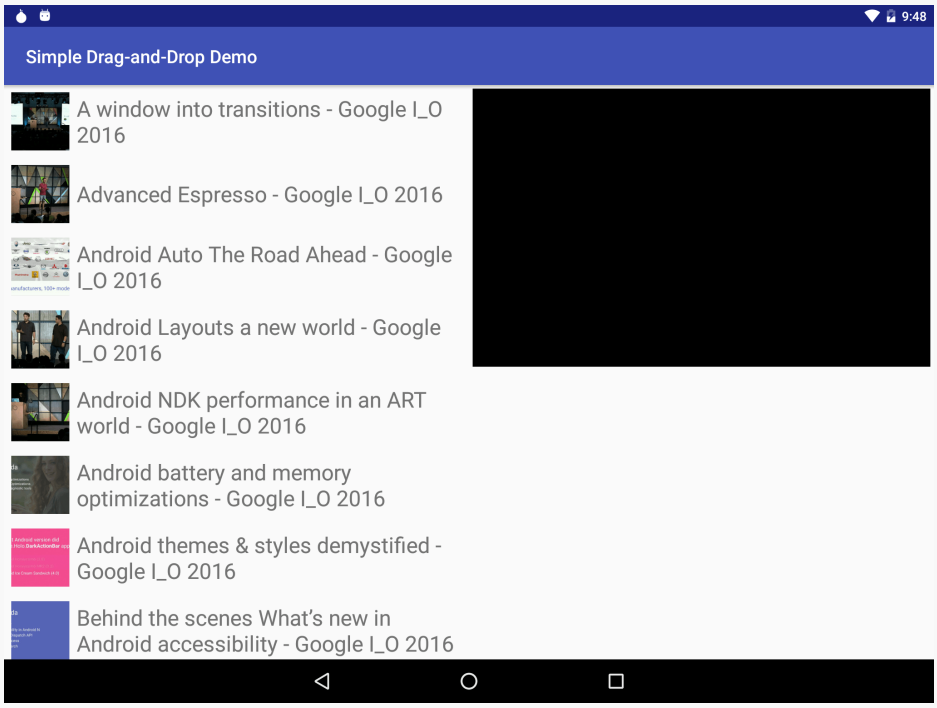


Figure 707: Simple Drag-and-Drop Demo, As Initially Launched

Once the user long-taps on a list row, a shadow based on the thumbnail appears under the user's finger, and the two drop targets show their red dashed outlines:

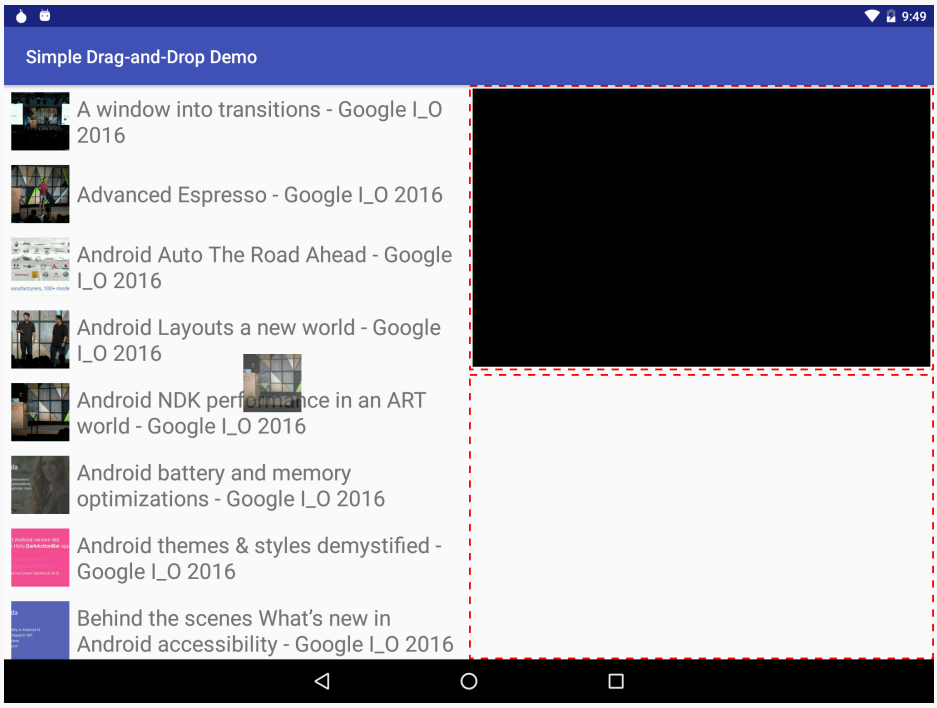


Figure 708: Simple Drag-and-Drop Demo, After Drag Started

If the user drags the drop shadow over one of the drop targets, it gets the `ACTION_DRAG_ENTERED` event and changes its outline to the green solid line:

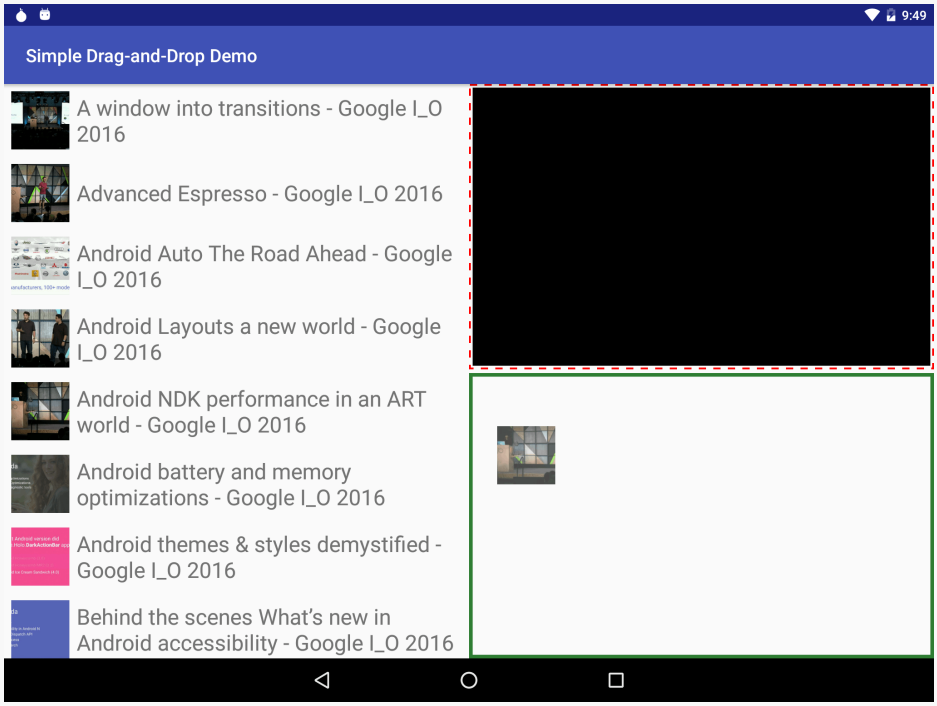


Figure 709: Simple Drag-and-Drop Demo, After Drag Enters Drop Target

Finally, if the user drops the item in one of the drop targets, it receives the `ACTION_DROP` event and can actually use the content:

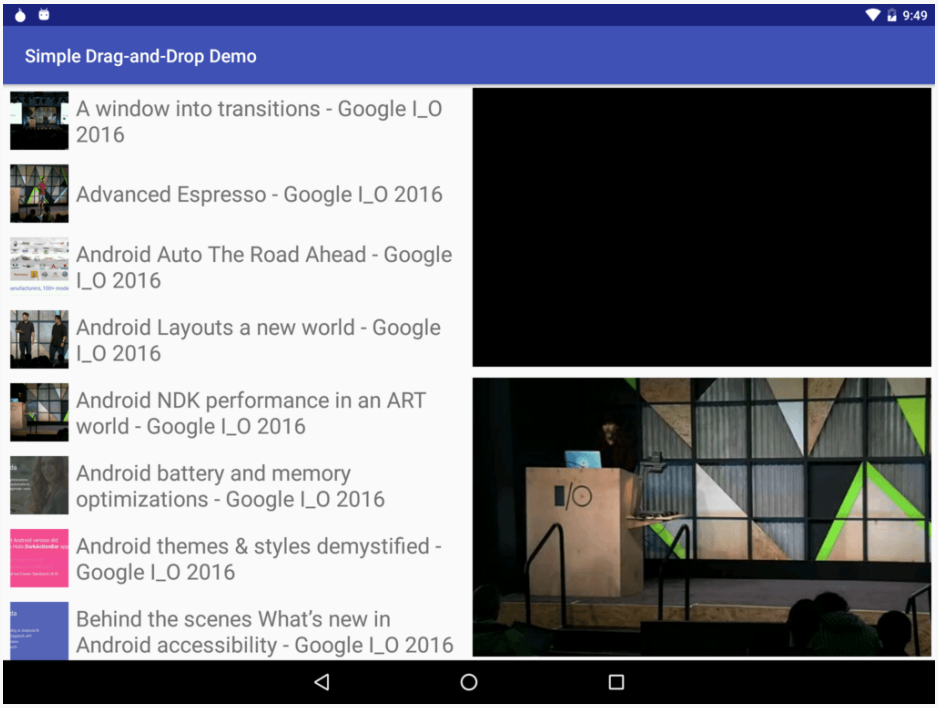


Figure 710: Simple Drag-and-Drop Demo, After Drop

Note that both widgets no longer show an outline, as they each received `ACTION_DROP_ENDED`, where they removed their outlines.

Drag-and-Drop, Between Apps

Android 7.0's multi-window capability ushers in a new era for drag-and-drop, where users drag-and-drop between apps. In theory, very little has to change to support drag-and-drop between apps.

However, there are challenges, the biggest one being permissions. The app with the drop target needs permission to work with whatever content is represented in the drag-and-drop operation. If that content is simply some plain text or something else that can be stuffed into a `ClipData`, permissions are part of drag-and-drop processing, as only the drop target selected by the user gets the `ACTION_DROP` event and can access that `ClipData`.

However, if the `ClipData` contains one or more `Uri` values, the app with the drop target needs access to that underlying content, just as it needs it for the clipboard or any other situation where a `Uri` is passed between apps.

The DragDrop/Permissions sample project demonstrates dragging and dropping between apps. This project has two app modules: drag and drop. As you might imagine, drag contains an activity that allows the user to drag something (in this case, an image), while drop contains an activity that accepts an image Uri and displays it.

Because cross-app drag-and-drop requires Android 7.0, both modules are set up with 7.0-compatible build settings in build.gradle — we will see the drag/build.gradle file shortly.

The Drag App

The drag app has a very simple UI: a single ImageView, set to fill the available space:

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView android:id="@+id/asset"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:scaleType="fitCenter" />
```

(from DragDrop/Permissions/drag/src/main/res/layout/main.xml)

The idea is that the user will long-click the ImageView to start the drag-and-drop operation. Hence, this is reminiscent of the DragDrop/Simple app, just with a single image, rather than one per row in a list.

The Custom Shadow

One problem with using the ImageView as the way to start the drag-and-drop operation comes with the drag shadow. In DragDrop/Simple, we used the image as the drag shadow. This worked well, because the image was a thumbnail, which usually is a good size for a drag shadow. In the drag app in DragDrop/Permissions, though, the ImageView is huge, far too large to use as the drag shadow. As a result, we cannot use the ImageView directly as before, but instead need to create a custom View.DragShadowBuilder subclass, named ThumbDragShadow:

```
  private class ThumbDragShadow extends View.DragShadowBuilder {
    @Override
    public void onProvideShadowMetrics(Point shadowSize,
                                       Point shadowTouchPoint) {
      shadowSize.set(iv.getWidth()/8, iv.getHeight()/8);
      shadowTouchPoint.set(shadowSize.x/2, shadowSize.y/2);
```

```
    }

    @Override
    public void onDrawShadow(Canvas canvas) {
      iv.draw(canvas);
    }
  }
```

(from DragDrop/Permissions/drag/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

This is a nested class inside `MainActivity`, and so it has access to the fields of `MainActivity`, such as our `ImageView`, named `iv`.

In `onProvideShadowMetrics()`, we set the size of the shadow to be 1/8th of the size of the `ImageView`. This is a sloppy approach and may wind up with too small of an image on smaller-screen devices. However, it does keep the aspect ratio of the `ImageView`. In addition, we set the touch point to be in the middle of the image — based on some Google sample code, it appears that this is a reasonable algorithm.

In `onDrawShadow()`, we need to draw something on the supplied `Canvas` that represents the drag shadow. In this case, we ask the `ImageView` to draw itself into that `Canvas`. This results in a cropped image, as the `ImageView` is much larger than our `Canvas`, which is sized based on the `Point` values we populated in `onProvideShadowMetrics()`. A better implementation would work with a `Bitmap` and scale it so the entire image would be seen in the drag shadow; this approach is used here for simplicity.

The StreamProvider

The image itself is stored in `assets/`. The photo is of One World Trade Center (a.k.a., "Freedom Tower") in New York City.

The reason for storing it in `assets/` is that not only do *we* need the image, but we need to provide other apps with access to the image. In this app, we will handle that using `StreamProvider`, from the author's CWAC-Provider library, as described in one of the chapters on the `ContentProvider` component.

To that end, we include the `cwac-provider` artifact in our `drag/build.gradle` file:

```
apply plugin: 'com.android.application'

dependencies {
    compile 'com.android.support:recyclerview-v7:25.3.1'
```

```
    compile 'com.squareup.picasso:picasso:2.5.2'
}

android {
    compileSdkVersion 24
    buildToolsVersion '26.0.2'

    defaultConfig {
        applicationId "com.commonsware.android.dragdrop.drag"
        minSdkVersion 24
        targetSdkVersion 24
    }

    aaptOptions {
        noCompress 'jpg'
    }
}

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:provider:0.4.0'
}
```

(from DragDrop/Permissions/drag/build.gradle)

Also note that we take steps to ensure that the build tools do not try to compress the JPEG further, by excluding `jpg` files from `aapt` compression via `noCompress` in `aaptOptions`

The manifest contains a `<provider>` element for our `StreamProvider`:

```
    <provider
      android:name="com.commonsware.cwac.provider.StreamProvider"
      android:authorities="${applicationId}.provider"
      android:exported="false"
      android:grantUriPermissions="true">
      <meta-data
        android:name="com.commonsware.cwac.provider.STREAM_PROVIDER_PATHS"
        android:resource="@xml/provider"/>
      <meta-data
        android:name="com.commonsware.cwac.provider.USE_LEGACY_CURSOR_WRAPPER"
```

```
        android:value="true"/>
    </provider>
```

(from DragDrop/Permissions/drag/src/main/AndroidManifest.xml)

That sets up the authority string to be the application ID with `.provider` appended. It also points `StreamProvider` to some XML metadata in `res/xml/provider.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>

  <asset name="assets" />

</paths>
```

(from DragDrop/Permissions/drag/src/main/res/xml/provider.xml)

Here, we say that we are willing to serve anything from `assets/`.

The Drag Request

In `onCreate()`, we use Picasso to load the image out of `assets/` and display it. However, we also register a `Callback` to find out when that has been completed:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    iv=(ImageView)findViewById(R.id.asset);

    Picasso.with(this)
      .load("file:///android_asset/FreedomTower-Morning.jpg")
      .fit().centerCrop()
      .into(iv, new Callback() {
        @Override
        public void onSuccess() {
          iv.setOnLongClickListener(MainActivity.this);
        }

        @Override
        public void onError() {
          // TODO
        }
      });
  }
```

(from DragDrop/Permissions/drag/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

We only call `setOnLongClickListener()` once the image has been loaded successfully, as until then, the user would not know what she is dragging and dropping.

Then, in `onLongClick()`, we start the drag-and-drop operation:

```
@Override
public boolean onLongClick(View view) {
  Uri uri=PROVIDER
    .buildUpon()
    .appendEncodedPath(StreamProvider.getUriPrefix(AUTHORITY))
    .appendEncodedPath("assets/FreedomTower-Morning.jpg")
    .build();

  ClipData clip=ClipData.newRawUri(getString(R.string.msg_photo), uri);
  View.DragShadowBuilder shadow=new ThumbDragShadow();

  iv.startDragAndDrop(clip, shadow, Boolean.TRUE,
    View.DRAG_FLAG_GLOBAL|View.DRAG_FLAG_GLOBAL_URI_READ|
      View.DRAG_FLAG_GLOBAL_PERSISTABLE_URI_PERMISSION);

  return(true);
}
```

(from DragDrop/Permissions/drag/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

First, we need a `Uri` pointing to our asset. We build such a `Uri` from:

- A static `PROVIDER` `Uri`, which incorporates our authority string:

```
private static final String AUTHORITY=
  BuildConfig.APPLICATION_ID+".provider";
private static final Uri PROVIDER=
  Uri.parse("content://"+AUTHORITY);
```

(from DragDrop/Permissions/drag/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

- The unique prefix used for this app by `StreamProvider` (via `getUriPrefix()`)
- The path to our asset

We then build a `ClipData` from that `Uri`, plus a string pulled from a resource. Note that it is unclear where this string is used, though accessibility options is one likely candidate.

The drag shadow is an instance of the `ThumbDragShadow` shown above.

To start the drag-and-drop operation, we call `startDragAndDrop()`. This is simply a new name for the `startDrag()` method. `startDrag()` is marked as deprecated in Android 7.0, replaced with `startDragAndDrop()`. However, for older, in-app drag-and-drop, feel free to use `startDrag()`, as it is your only option for Android 6.0 and older devices.

This time, we pass in some flags:

- `DRAG_FLAG_GLOBAL` indicates that we want the drag-and-drop operation to work between apps. If we left the flags as 0, the drag-and-drop would be limited only to this app. In this respect, a drag is local by default, with cross-app drag-and-drop being something you have to explicitly opt into.
- `DRAG_FLAG_GLOBAL_URI_READ` indicates that we want the other app to be able to read the content identified by the `Uri` that we are putting into the `ClipData`. Without this, any app receiving the `DragEvent` would be unable to display the image. Note that there is an equivalent `DRAG_FLAG_GLOBAL_URI_WRITE` if you want to offer write access.
- `DRAG_FLAG_GLOBAL_PERSISTABLE_URI_PERMISSION` indicates that we want to grant the recipient app durable rights to the content identified by the `Uri` that we are putting into the `ClipData`. The term "persistable", and the documentation for this flag, suggests that this access survives reboots. That may be excessive here. We will explore why we are using this flag when we look at the `drop` app.

The Drop App

The `drop` app is a version of the drop logic from `DragDrop/Simple`, reduced to just handling the drop in an `ImageView`. However, it does have a few wrinkles, both related to drag-and-drop (permissions) and related to general Android development (configuration change support).

The Layout

The revised layout is simply the `ImageView`, wrapped in the `FrameLayout` for the drop hint drawables:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout android:id="@+id/thumbnail_frame"
  xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_margin="4dp"
  android:padding="4dp">

  <ImageView
    android:id="@+id/thumbnail_large"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="centerInside" />
</FrameLayout>
```

(from DragDrop/Permissions/drop/src/main/res/layout/main.xml)

In `onCreate()` and `onSaveInstanceState()`, we load that layout, get the `ImageView`, and populate it (via the same `showThumbnail()` as before) if we have a `thumbnailUri` from our saved instance state `Bundle`:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    image=(ImageView)findViewById(R.id.thumbnail_large);
    image.setOnDragListener(this);

    if (state!=null) {
      imageUri=state.getParcelable(STATE_IMAGE_URI);

      if (imageUri!=null) {
        showThumbnail();
      }
    }
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putParcelable(STATE_IMAGE_URI, imageUri);
  }
```

(from DragDrop/Permissions/drop/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

The Drag Event

The `onDrag()` method is the same as before, except for two events: `ACTION_DRAG_STARTED` and `ACTION_DROP`.

We are expecting to get a `Uri` pointing to an image from the outside app via a drag-and-drop operation. Ideally, we would validate that in `ACTION_DRAG_STARTED`, returning `false` if the content is something else:

```
@Override
public boolean onDrag(View v, DragEvent event) {
  boolean result=true;

  switch (event.getAction()) {
    case DragEvent.ACTION_DRAG_STARTED:
      if (event
        .getClipDescription()
        .hasMimeType(ClipDescription.MIMETYPE_TEXT_URILIST)) {
        applyDropHint(v, R.drawable.droppable);
      }
      else {
        result=false;
      }

      break;

    case DragEvent.ACTION_DRAG_ENTERED:
      applyDropHint(v, R.drawable.drop);
      break;

    case DragEvent.ACTION_DRAG_EXITED:
      applyDropHint(v, R.drawable.droppable);
      break;

    case DragEvent.ACTION_DRAG_ENDED:
      applyDropHint(v, -1);
      break;

    case DragEvent.ACTION_DROP:
      requestDragAndDropPermissions(event);

      ClipData.Item clip=event.getClipData().getItemAt(0);

      imageUri=clip.getUri();
      showThumbnail();
      break;
```

```
    }

    return(result);
  }
```

(from DragDrop/Permissions/drop/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

Unfortunately, all we can do is determine that we are getting *some* `Uri`. The MIME type in our `ClipDescription` is *not* the MIME type of the content underlying our `Uri`, but rather will be `ClipDescription.MIMETYPE_TEXT_URILIST`. This is because a `ClipData` can have several items, each with `Uri` values. We have no way, given just the `ClipDescription` to determine if we actually have an image `Uri`. So, as long as we are getting a `Uri` value, we assume that the drop might be meaningful and return `true`.

For `ACTION_DROP`, we first call `requestDragAndDropPermissions()`, to grant our app the rights offered to us by whatever app initiated the drag-and-drop operation:

```
      case DragEvent.ACTION_DROP:
        requestDragAndDropPermissions(event);

        ClipData.Item clip=event.getClipData().getItemAt(0);

        imageUri=clip.getUri();
        showThumbnail();
        break;
```

(from DragDrop/Permissions/drop/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

The `requestDragAndDropPermissions()` method returns a `DragAndDropPermissions` object. The JavaDocs for this class point out the lifetime of our permissions:

> The life cycle of the permissions is bound to the activity used to call `requestDragAndDropPermissions()`. The permissions are revoked when this activity is destroyed, or when `release()` is called, whichever occurs first.

However, the user could destroy your activity at any point in time, via a configuration change. As a result, you have three main options here:

1. Make a local copy of the content as soon as you get the `Uri`, hopefully before your activity gets destroyed via a configuration change
2. Opt out of the automatic destroy-and-recreate cycle for configuration changes for any activity that has drop targets, via `android:configChanges` in the manifest, and deal with all the problems that technique raises

3. Ignore the issue and hope that the app that started the drag-and-drop operation included `DRAG_FLAG_GLOBAL_PERSISTABLE_URI_PERMISSION` in its `startDragAndDrop()` call

But, after the `requestDragAndDropPermissions()` call, we grab the first `Uri` out of the `ClipData`, store that in the `imageUri` field, and have `showThumbnail()` display that image via Picasso. A better approach would examine each possible `Uri` in the `ClipItem` for one that represents an image (showing that one), and returning `false` from `onDrag()` if no such `Uri` is found, so the drag-and-drop operation remains active.

The Results

If you run both apps, and have them both visible in a multi-window environment (e.g., split-screen mode on a phone or tablet), you will be able to drag and drop between them:

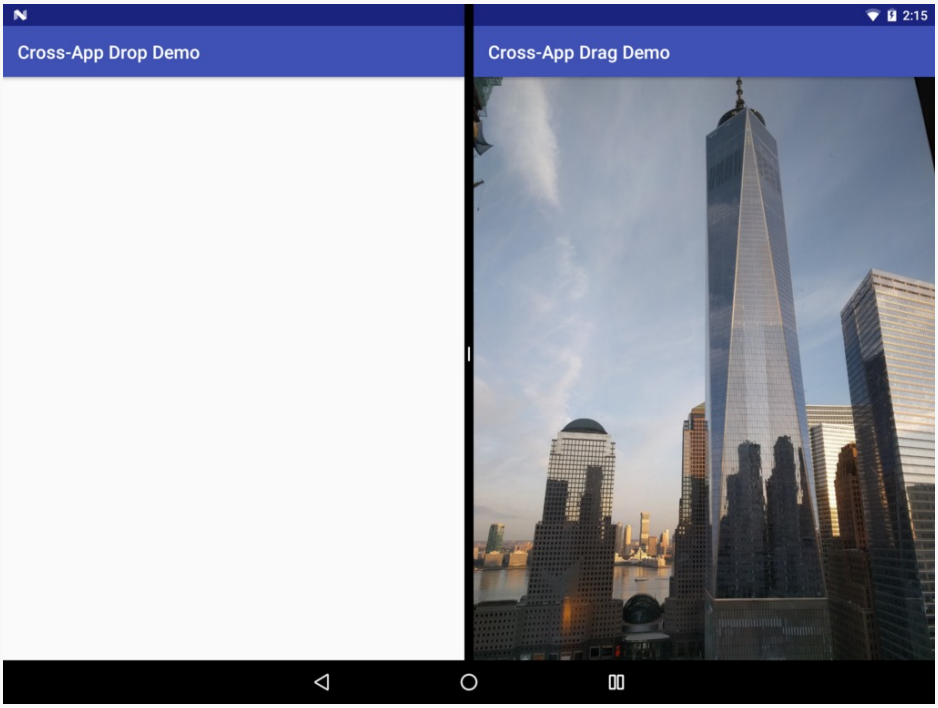


Figure 711: Cross-App Drag-And-Drop, Showing Both Activities As Initially Launched

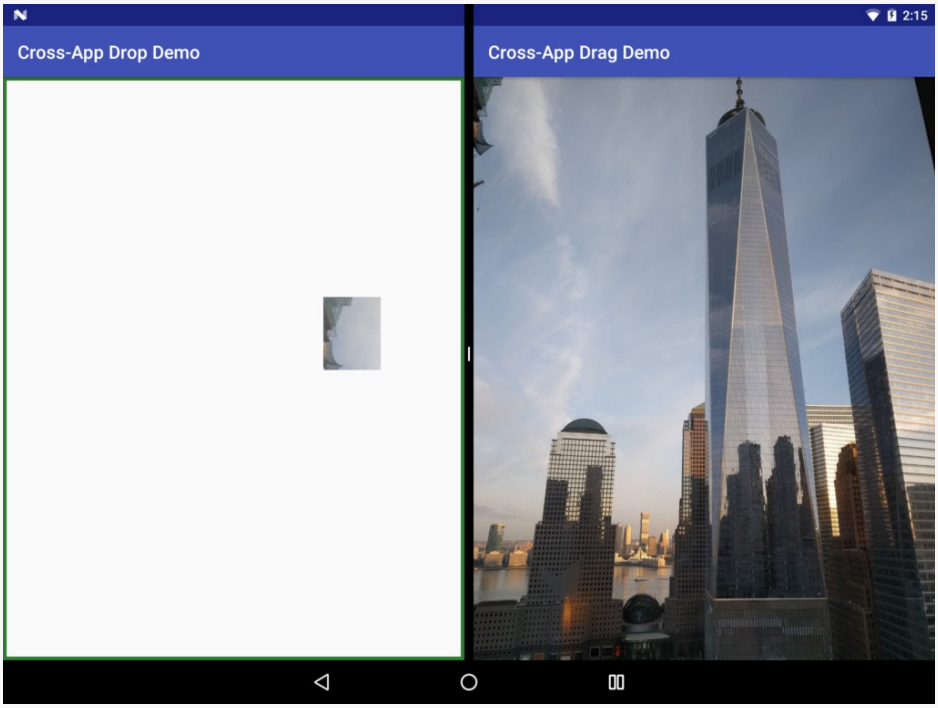


Figure 712: Cross-App Drag-And-Drop, Showing Drag Shadow

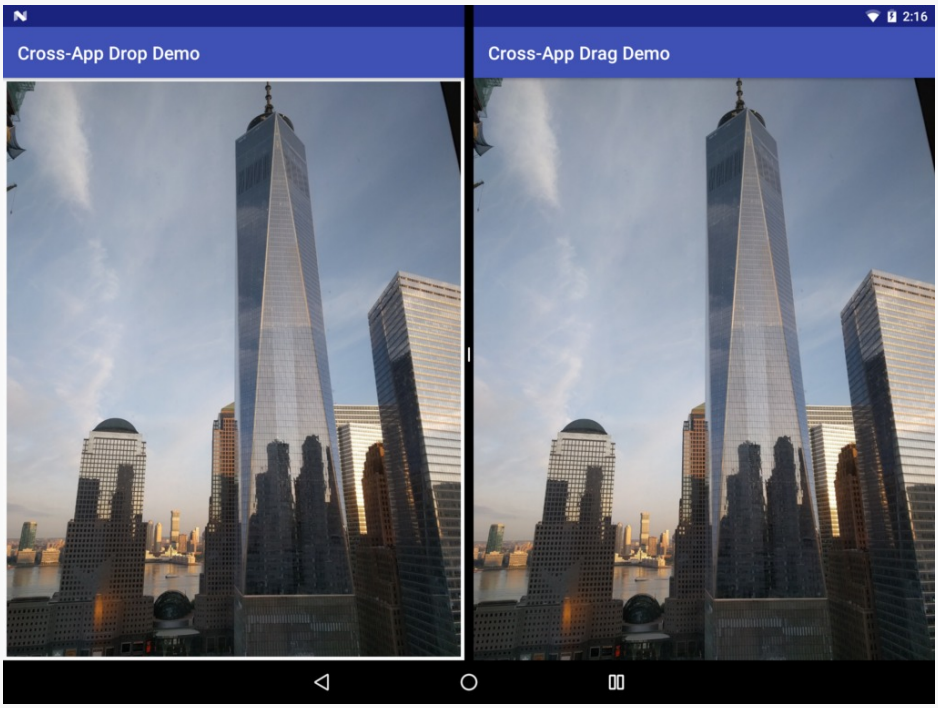


Figure 713: Cross-App Drag-And-Drop, Showing Result of Drag-and-Drop

Detecting Cross-App Drag Events

In the `DragDrop/Permissions` sample, there is nothing in our `onDrag()` method, or anywhere else, that indicates that we want to allow drag events from third-party apps. **That happens by default, and there is no way to stop it**. Hence, *any* app implementing official drag-and-drop support *has* to support arbitrary apps passing in content. With luck, this too will get changed.

We were given this recipe for detecting cross-app drag-and-drop:

- Fill in a non-`null` value for the local state in the call to `startDragAndDrop()`
- Check in `ACTION_DRAG_STARTED` to see if the local state is `null`, in which case, the drag-and-drop must have started from some other app

However, this does not work well. The local state is local to a *window*, not an app or process. As a result, if your app is visible in more than one window — for example, you used `FLAG_ACTIVITY_LAUNCH_ADJACENT` to start up another activity in another window — then you will lose the local state even for in-app drag-and-drop across these windows.

In the `Simple` drag-and-drop sample from earlier, we check to see if the local state is `null` and reject the drag event if it is. There, we are not expecting to have activities in multiple windows, so all drag-and-drop work should be local. Conversely, we do *not* check the local state in the `Permissions` sample, where we specifically *want* cross-app drag-and-drop.

Intra-App Cross-Window Drag-and-Drop

Drag-and-drop not only works between apps with Android 7.0's multi-window feature — it also works for two windows within the same app.

The `DragDrop/SplitScreen` sample project is a clone of the `DragDrop/Permissions` project. However, both the drag activity (`MainActivity`) and the drop activity (`DropActivity`) are in the same app module (`app/`).

Also, `MainActivity` now has an action bar with a "launch" item that, when tapped, will bring up the `DropActivity` in an adjacent window:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
```

```
    if (item.getItemId()==R.id.launch) {
      startActivity(new Intent(this, DropActivity.class)
        .setFlags(Intent.FLAG_ACTIVITY_LAUNCH_ADJACENT |
                  Intent.FLAG_ACTIVITY_NEW_TASK |
                  Intent.FLAG_ACTIVITY_MULTIPLE_TASK));

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from DragDrop/SplitScreen/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

Launching `MainActivity`, entering multi-window mode, and launching `DropActivity` gives you those two activities in separate windows, such as in split-screen mode:

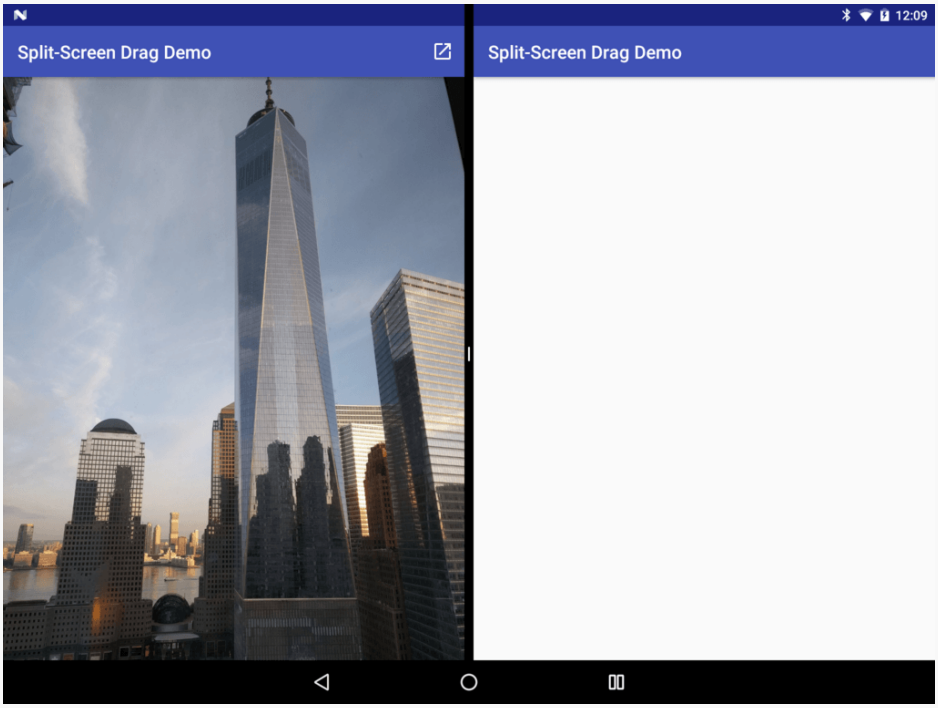


Figure 714: Split-Screen Drag Demo, With Both Activities Showing

And, as in the cross-app scenario, you can drag from one window into the other:

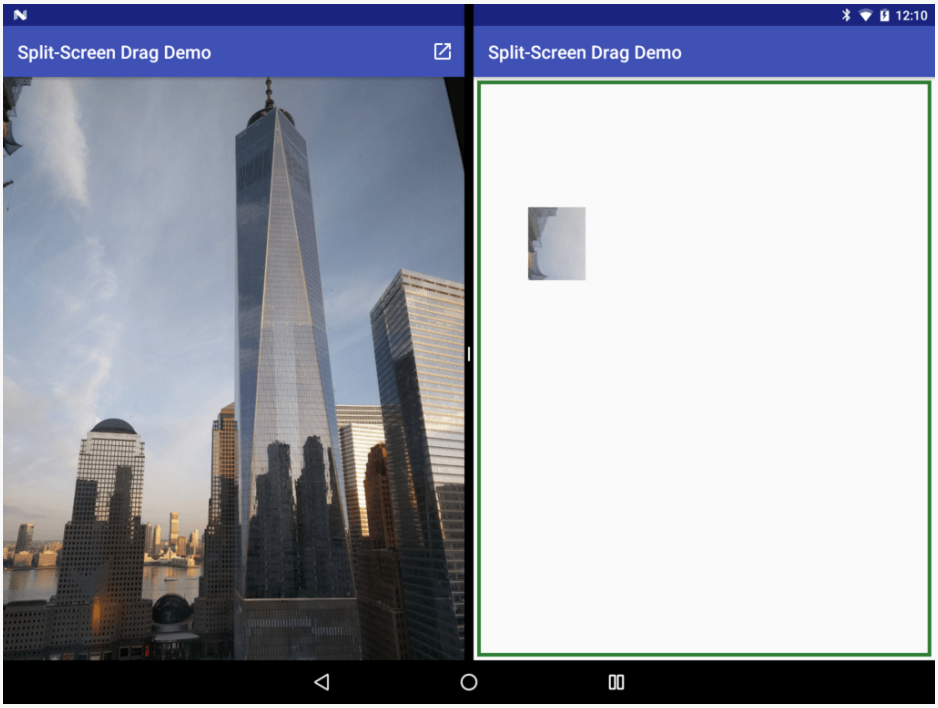


Figure 715: Split-Screen Drag Demo, Part-Way Through a Drag

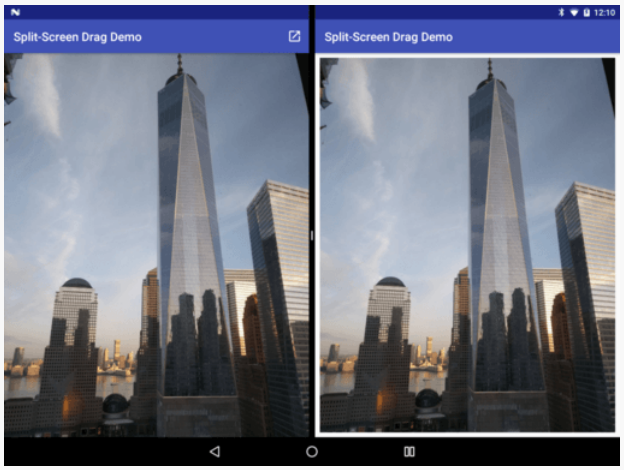


Figure 716: Split-Screen Drag Demo, After Drag-and-Drop

However, as was noted above, the local state data is lost in these cross-window drag-and-drop operations, making them indistinguishable from cross-app drag-and-drop operations.

Pondering Legacy Multi-Window

In principle, cross-app drag-and-drop could work with Samsung's legacy multi-window support. However, calling `startDrag()` (the pre-Android 7.0 equivalent of `startDragAndDrop()`) will not grant permissions to the other app to use the content associated with any `Uri`. The only way to make this work would be if the content identified by the `Uri` was readable by all apps, which is not great from a security standpoint.

LG's legacy multi-window support does not seem to support cross-app drag-and-drop.

Also, Chrome OS does not support cross-app drag-and-drop as of the August 2016 round of developer previews.

Dragging and Dropping Simple Stuff

This chapter focuses on drag-and-drop of content, represented by a `Uri`. That is not your only option, just as that is not your only option for putting stuff onto, or removing stuff from, the clipboard.

A `ClipData` object contains one or more `ClipData.Item` objects. These can be of three main forms:

- Text, in the form of a `CharSequence`, including support for any standard Android spans
- An `Intent`, usually designed for creating some sort of shortcut to be able to launch an activity identified by the `Intent`
- A `Uri`

Outside of specialized cases (e.g., home screens), if you are not using `Uri`, probably you are using text.

If you are implementing a drop target, and all you know how to do is handle text, you can call `coerceToText()` on a `ClipData.Item` object to get the best text representation of whatever it is. For `Uri` values pointing to text content, `coerceToText()` will read in the content and return it. For anything else, you get back `toString()` on the content, more or less.

Multi-Action Drag-and-Drop

The previous section brings up home screens as an example of drag-and-drop. On many Android home screen implementations, if you long-click on an icon in the launcher, you can drag-and-drop that icon into the home screen itself, thereby creating a shortcut. However, in addition to that, many home screens also offer special drop targets tied to specific actions, such as "Uninstall". If the user drops the icon over the home screen area, a shortcut gets created; if the user drops the icon over a special drop target, the action for that target is performed.

You can do this too.

Mostly, it is a matter of arranging to show those special drop targets only during a drag-and-drop operation, then handling those drops specifically. However, due to the nature of Android's view hierarchy and the drag-and-drop framework, you need to ensure that you show the special drop target's view *before* you start the drag-and-drop operation. Otherwise, the special drop target's view will never receive `ACTION_DRAG_ENTERED` or `ACTION_DROP` events.

We can see how this works in practice in the `DragDrop/Action` sample project. This is a clone of the `DragDrop/Simple` app from earlier in this chapter, except that we have added a special "Video Info" drop target.

The Layout

The `-w800dp` layout is mostly as it was in the original app, except that we have added an `info` `TextView` above the `RecyclerView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">

  <LinearLayout
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:orientation="vertical">

    <TextView
      android:id="@+id/info"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_margin="8dp"
    android:padding="8dp"
    android:text="@string/label_video_info"
    android:textAppearance="?android:textAppearanceLarge"
    android:visibility="gone" />

  <android.support.v7.widget.RecyclerView
    android:id="@+id/video_list"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
</LinearLayout>

<LinearLayout
  android:layout_width="0dp"
  android:layout_height="match_parent"
  android:layout_weight="1"
  android:orientation="vertical">

  <FrameLayout
    android:id="@+id/video_frame"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_marginBottom="4dp"
    android:layout_weight="1"
    android:padding="4dp">

    <VideoView
      android:id="@+id/player"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_gravity="center" />
  </FrameLayout>

  <FrameLayout
    android:id="@+id/thumbnail_frame"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:padding="4dp">

    <ImageView
      android:id="@+id/thumbnail_large"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:scaleType="centerInside" />
```

```
    </FrameLayout>
  </LinearLayout>
</LinearLayout>
```

(from DragDrop/Action/app/src/main/res/layout-w8oodp/main.xml)

However, this `TextView` has a `visibility` of gone at the outset, so it will not show up for users.

Showing and Hiding the Action

`onCreate()` initializes an `info` field with the `TextView`, much as it initializes the fields for the `VideoView` and `ImageView`:

```
    player=(VideoView)findViewById(R.id.player);

    if (player!=null) {
      player.setOnDragListener(this);
    }

    thumbnailLarge=(ImageView)findViewById(R.id.thumbnail_large);

    if (thumbnailLarge!=null) {
      thumbnailLarge.setOnDragListener(this);
    }

    info=findViewById(R.id.info);

    if (info!=null) {
      info.setOnDragListener(this);
    }
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

However, since `info` is gone, the user cannot drag anything over it. We need to arrange to make it visible.

You might try making it visible in `onDrag()`, when we get the `ACTION_DRAG_STARTED` event. After all, this would seem to describe what we want: when the drag begins, show the special drop target as an option for the user. Unfortunately, this does not work: while the user can see the `info` `TextView`, that `TextView` does not get any further `onDrag()` events.

Instead, we are forced to arrange to make the `info` view visible before starting the drag-and-drop operation. As a side effect, that requirement means that we cannot

use this technique for cross-app drag-and-drop, since we have no idea when some other app starts the drag-and-drop operation.

In fact, it is even a bit awkward to handle in this app, as it is the `RowController` that initiates the drag-and-drop operation. The `RowController` knows nothing about the `info` view, nor should it. Instead, we need to have the `RowController` let the `MainActivity` know that a drag-and-drop operation is about to start, so the activity can show the view.

To that end, `RowController` defines an `OnStartDragListener` interface. It expects to get such a listener as a constructor parameter, storing it in a field for later use:

```
class RowController extends RecyclerView.ViewHolder
    implements View.OnClickListener, View.OnLongClickListener {
  interface OnStartDragListener {
    void onStartDrag();
  }

  final private TextView title;
  final private ImageView thumbnail;
  private Uri videoUri=null;
  private String videoMimeType=null;
  final private OnStartDragListener listener;

  RowController(View row, OnStartDragListener listener) {
    super(row);

    this.listener=listener;
    title=(TextView)row.findViewById(android.R.id.text1);
    thumbnail=(ImageView)row.findViewById(R.id.thumbnail);

    row.setOnClickListener(this);
    row.setOnLongClickListener(this);
  }
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/RowController.java)

Then, in `onLongClick()`, if we have a listener, we call `onStartDrag()` on it, to indicate that we are about to start a drag-and-drop operation:

```
  @Override
  public boolean onLongClick(View v) {
    if (listener!=null) {
      listener.onStartDrag();
    }
```

```
    ClipData clip=ClipData.newRawUri(title.getText(), videoUri);
    View.DragShadowBuilder shadow=new View.DragShadowBuilder(thumbnail);

    itemView.startDrag(clip, shadow, Boolean.TRUE, 0);

    return(true);
  }
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/RowController.java)

The `RowController` instances are created by the `VideoAdapter`. Fortunately, `VideoAdapter` is a nested class inside of `MainActivity`. So, we implement `OnStartDragListener` on `MainActivity` and pass the activity instance to the `RowController` constructor:

```
    @Override
    public RowController onCreateViewHolder(ViewGroup parent,
                                            int viewType) {
      return(new RowController(getLayoutInflater()
        .inflate(R.layout.row, parent, false), MainActivity.this));
    }
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

All the `onStartDrag()` method does is make the `info` view visible:

```
  public void onStartDrag() {
    info.setVisibility(View.VISIBLE);
  }
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

If we make it visible, clearly we need to hide it again at some future point. That would be when the drag-and-drop operation has completed, and it is safe for us to mark `info` as `GONE` in our handling of `ACTION_DRAG_ENDED` in `onDrag()`:

```
      case DragEvent.ACTION_DRAG_ENDED:
        applyDropHint(v, -1);
        info.setVisibility(View.GONE);
        break;
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

Handling Drag Events

We still want to show the drop hint backgrounds, but in this case, we apply them directly to the `TextView`, rather than going with a wrapping `FrameLayout`. So, we adjust `applyDropHint()` to only work with the parent of the view if this is not the `info` view:

```
private void applyDropHint(View v, int drawableId) {
  if (v!=info) {
    v=(View)v.getParent();
  }

  if (drawableId>-1) {
    v.setBackgroundResource(drawableId);
  }
  else {
    v.setBackground(null);
  }
}
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

And, in `ACTION_DROP` processing, if the user dropped the content over the `info` view, we simply show a `Toast` with the text of the `Uri`:

```
    case DragEvent.ACTION_DROP:
      ClipData.Item clip=event.getClipData().getItemAt(0);
      Uri videoUri=clip.getUri();

      if (v==player) {
        player.setVideoURI(videoUri);
        player.start();
      }
      else if (v==info) {
        Toast
          .makeText(this, videoUri.toString(), Toast.LENGTH_SHORT)
          .show();
      }
      else {
        Picasso.with(thumbnailLarge.getContext())
          .load(videoUri.toString())
          .fit().centerCrop()
          .placeholder(R.drawable.ic_media_video_poster)
          .into(thumbnailLarge);
      }
```

(from DragDrop/Action/app/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

The Result

Now, when you run the app on a sufficiently-wide screen, and you start a drag-and-drop operation, the "Video Info" `TextView` appears and serves as a drop target:

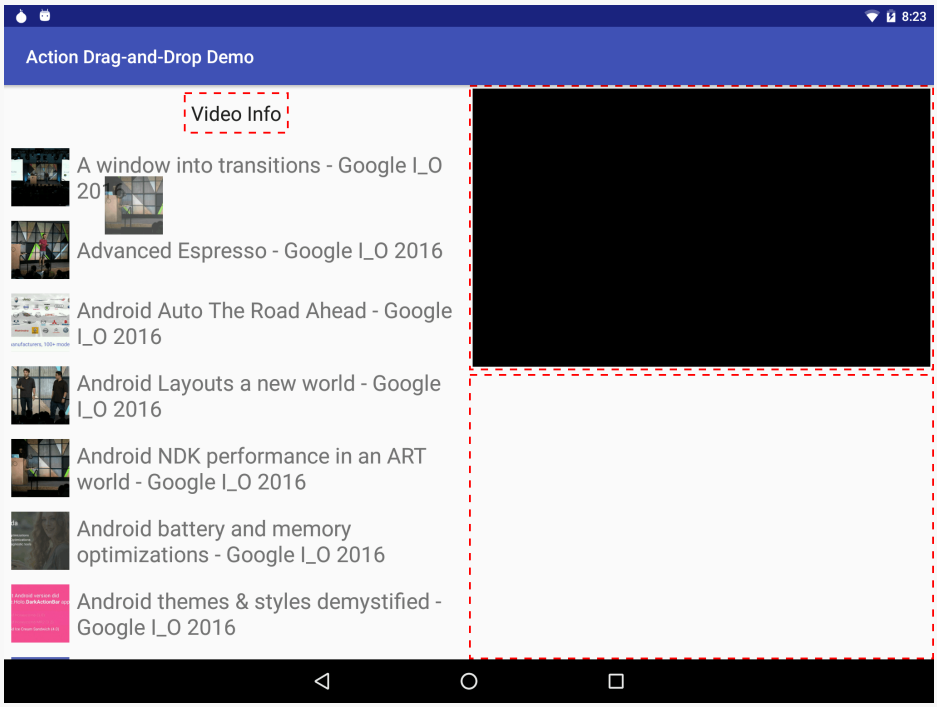


Figure 717: Drag-And-Drop, Showing Special Drop Target

Nested Drop Targets

You may have a need for nested drop targets, where you are listening for drag events both on some container and on some view inside of that container. For example, if you are using the drag-and-drop APIs to support reordering items in a `LinearLayout` or `RecyclerView`, there is a good chance that you will need to have listeners both on that container and on existing items in the container (e.g., to animate them out of the way to allow the user to drop in the newly-vacant spot).

Alas, this is an area that has some undocumented behavior changes in Android 7.0, as Dan Lew uncovered in August 2016.

The `DragDrop/Nougat` sample project illustrates the old and new behavior, plus some code to get Android 7.0 to behave more like the older versions of Android.

The behavior change is triggered by nested drop targets, which means we need a nested layout, such as this one:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout android:id="@+id/outer_container"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/outer_normal"
  android:padding="48dp">

  <FrameLayout
    android:id="@+id/inner_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/inner_normal"
    android:padding="48dp">

    <ImageView
      android:id="@+id/thumbnail_large"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:background="@color/image_normal"
      android:contentDescription="@string/icon"
      android:scaleType="centerInside" />
  </FrameLayout>

</FrameLayout>
```

(from DragDrop/Nougat/app/src/main/res/layout/bug.xml)

Here, we have a `FrameLayout` holding onto another `FrameLayout`, which holds onto an `ImageView`, each with a different background color:

Figure 718: Garish Background Colors

Further, suppose that we are interested in drag events for all three of these, so we call `setOnDragListener()` for each of them. In the code, we set the background of a drop target to green when we get an `ACTION_DRAG_ENTERED` event for the target, and we revert to the original background in `ACTION_DRAG_EXITED`, `ACTION_DRAG_ENDED`, and `ACTION_DROP`.

The Behavior Prior to Android 7.0

On Android 6.0 and earlier, drag events are inclusive. In other words, if the user has dragged an item into the inner `FrameLayout`, this is *also* considered to be inside the outer `FrameLayout`. From an event standpoint, the outer `FrameLayout` only gets an `ACTION_DRAG_EXITED` event when the dragged item leaves its outer boundaries.

So, as the user drags an item into our nested drop targets, outer targets remain green, even as the item enters inner targets:

Figure 719: Dragging Over Outer Drop Target on Android 6.0

Figure 720: Dragging Over Inner Drop Target on Android 6.0

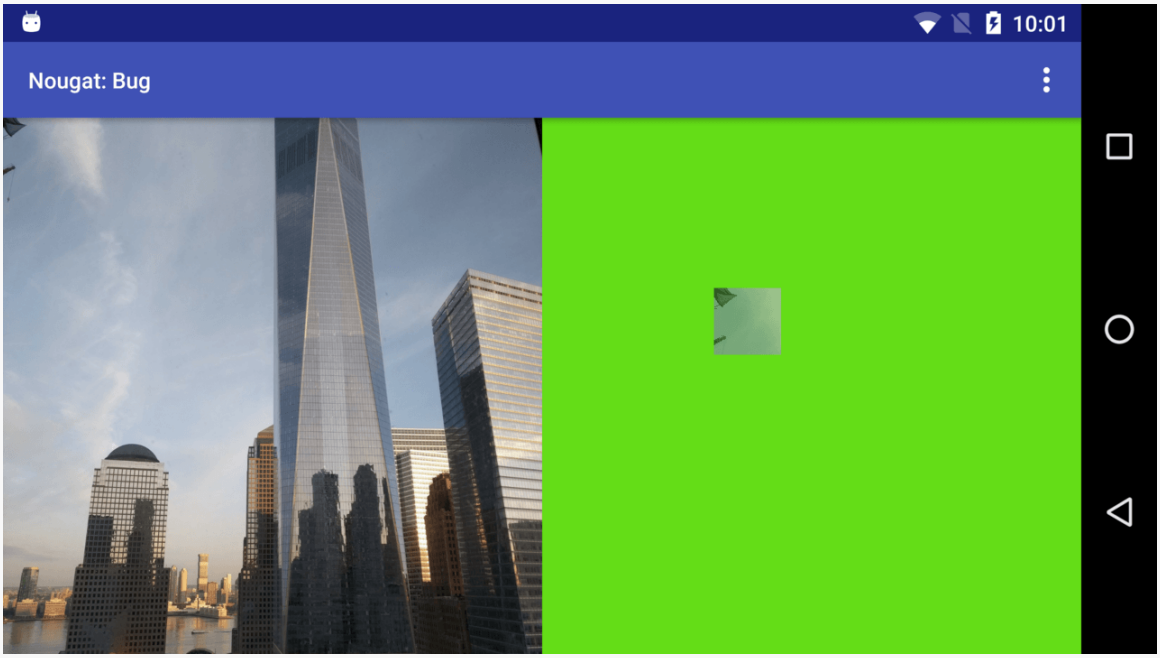


Figure 721: Dragging Over ImageView Drop Target on Android 6.0

Android 7.0 Behavior

However, on Android 7.0, drag events are exclusive. If the user drags an item into the inner `FrameLayout`, the item will *exit* the outer `FrameLayout` from a drag-and-drop perspective. So, as the user drags the item towards the inner-most drop target, the user exits the outer drop targets, and we restore the backgrounds along the way:

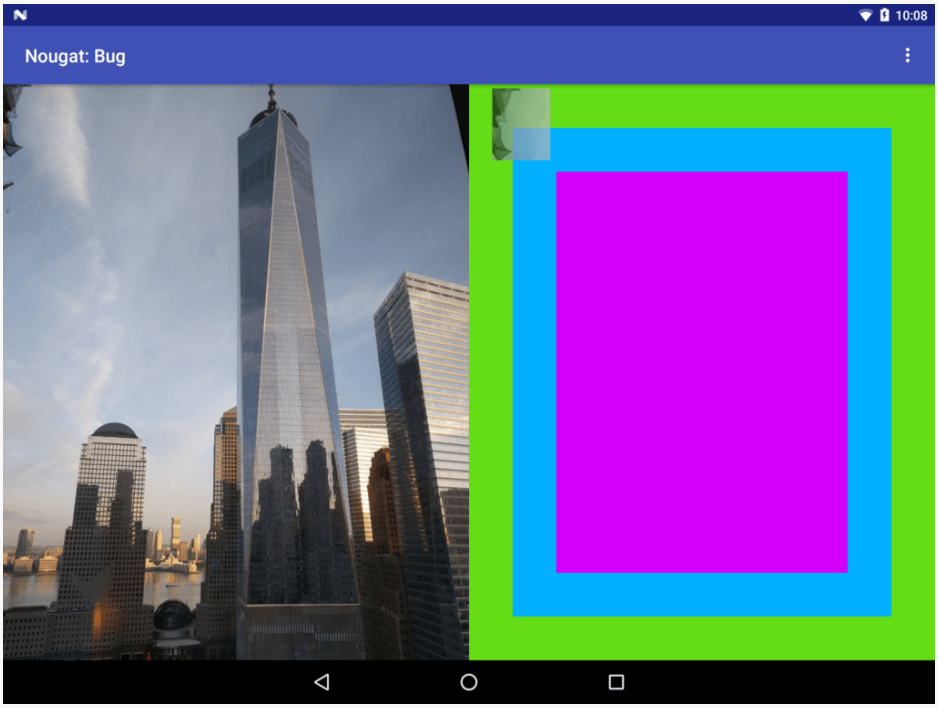


Figure 722: Dragging Over Outer Drop Target on Android 7.0

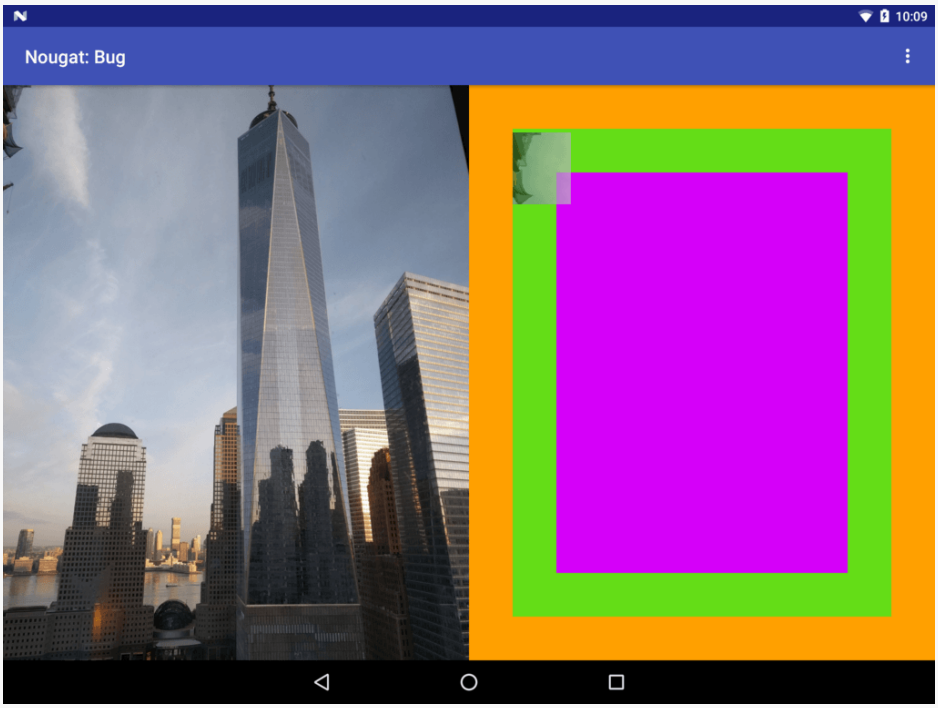


Figure 723: Dragging Over Inner Drop Target on Android 7.0

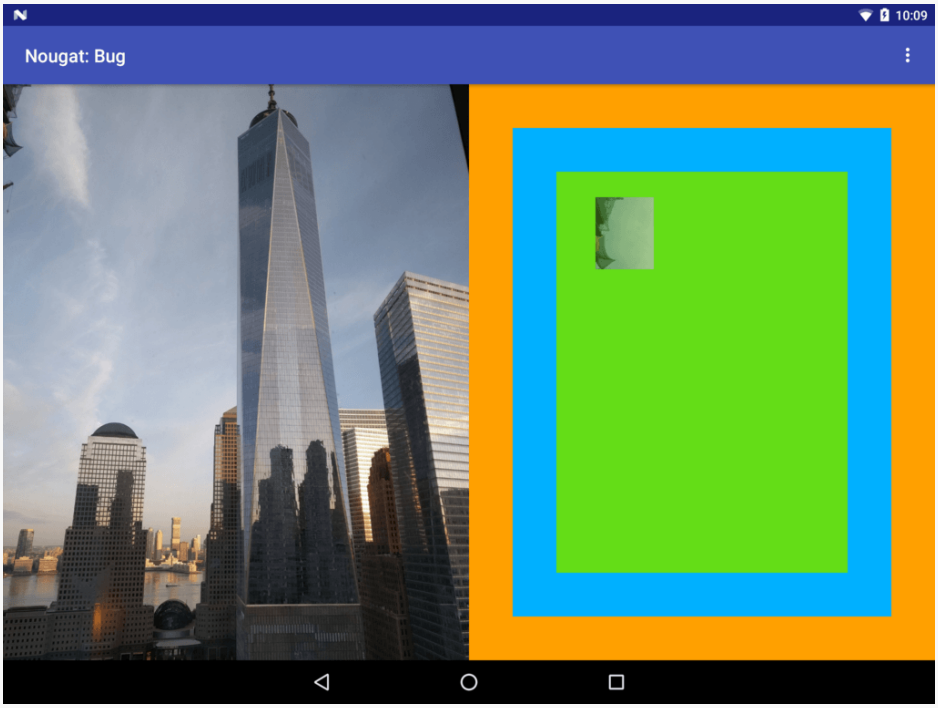


Figure 724: Dragging Over ImageView Drop Target on Android 7.0

Getting Inclusive on Android 7.0

Which approach is "correct" is somewhat immaterial, as we need consistency across Android OS versions.

The sample app offers one workaround for the discrepancy: a `DropTarget` class that provides inclusive behavior on Android 7.0. Basically, it serves as a composite `OnDragListener`, forwarding `onDrag()` calls to widgets as appropriate. This includes forwarding events to parents of widgets, if those parents are part of the `DropTarget` coverage:

```
package com.commonsware.android.dragdrop;

import android.os.Build;
import android.view.DragEvent;
import android.view.View;
import android.view.ViewParent;
import java.util.ArrayList;

public class DropTarget implements View.OnDragListener {
  private ArrayList<View> views=new ArrayList<>();
```

```
  private View.OnDragListener listener;

  public DropTarget on(View... views) {
    for (View v : views) {
      this.views.add(v);
      v.setOnDragListener(this);
    }

    return(this);
  }

  public void to(View.OnDragListener listener) {
    this.listener=listener;
  }

  @Override
  public boolean onDrag(View view, DragEvent dragEvent) {
    if (Build.VERSION.SDK_INT<Build.VERSION_CODES.N) {
      return(listener.onDrag(view, dragEvent));
    }

    boolean result=listener.onDrag(view, dragEvent);
    ViewParent parent=view.getParent();

    while (parent!=null && parent instanceof View) {
      View parentView=(View)parent;

      if (views.contains(parentView)) {
        listener.onDrag(parentView, dragEvent);
      }

      parent=parentView.getParent();
    }

    return(result);
  }
}
```

(from DragDrop/Nougat/app/src/main/java/com/commonsware/android/dragdrop/DropTarget.java)

For each of the `views` that `DropTarget` is set to monitor, the `DropTarget` sets itself up as the `OnDragListener`. In `onDrag()`, if we are on a pre-7.0 version of Android, `DropTarget` just forwards the event along normally. If, however, we are on Android 7.0 or higher, `onDrag()`:

- Calls `onDrag()` for the listener attached to the `DropTarget`, using the `View` originally passed into `onDrag()`, and holding onto the result `boolean`

- Walks the chain of parents, calling `onDrag()` on the listener for each of those that are part of our monitored set of `views`
- Returns the value from the original (inner-most) `onDrag()` call on the listener

Then, instead of registering the activity as the `OnDragListener` like this:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  image=(ImageView)findViewById(R.id.thumbnail_large);
  image.setOnDragListener(this);
  findViewById(R.id.outer_container).setOnDragListener(this);
  findViewById(R.id.inner_container).setOnDragListener(this);
}
```

(from DragDrop/Nougat/app/src/main/java/com/commonsware/android/dragdrop/BugActivity.java)

we can use `DropTarget`:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  image=(ImageView)findViewById(R.id.thumbnail_large);

  new DropTarget()
    .on(image,
      findViewById(R.id.outer_container),
      findViewById(R.id.inner_container))
    .to(this);
}
```

(from DragDrop/Nougat/app/src/main/java/com/commonsware/android/dragdrop/InclusiveActivity.java)

The result is more inclusive behavior. In the case of this sample app, it happens to be sufficient to allow Android 7.0 to behave as does Android 6.0 and earlier. More sophisticated drag-and-drop implementations may need a more sophisticated approach than `DropTarget` to achieve a similar workaround.

The State of the Bugs

Both bugs filed by Dan Lew have been marked as “FutureRelease”, with a comment on one indicating that the new exclusive behavior would be restricted to apps with a

`targetSdkVersion` of 24 or higher. It is unclear exactly when this change will be released. And, since many apps will have a `targetSdkVersion` of 24 or higher to avoid warning messages in Android 7.0+ multi-window, the proposed fixes may not help much.

Pondering Standards

Framework-supplied drag-and-drop has not been all that popular to date. As a result, there are no common conventions for how to designate drop targets for the user. Similarly, as of mid-2016, there appears to be no recommendations for this in the Material Design guidelines from Google.

Eventually, the Android development community will start to coalesce around certain patterns, with or without Google's assistance. Experiment now, but watch for conventions to emerge, then adopt those conventions, where they make sense for your app.

Pondering Accessibility

Drag-and-drop is not particularly accessible. Visually-impaired users may have difficulty discerning where one can drag from and where one can drag to. Motor-impaired users may have difficulty doing the gesture to initiate drag-and-drop or in dragging the shadow to the desired location.

As a result, while drag-and-drop is a worthy feature, ensure that it is not the *only* option for performing some action. There should be some other way to do that action, and perhaps more than one other way to do that action, such as:

- Keyboard shortcuts
- Copy-and-paste, perhaps using action modes
- Direct manipulation (e.g., use drag-and-drop on overview screens with lists, plus offer an action bar item or other affordances on detail screens to perform the same action)

Keyboard and Mouse Input

More and more Android users are starting to use external keyboards and mice with their devices. Sometimes, the device is designed for such use, such as the Jide Remix Mini or all-on-one units like the HP Slate 21. Some people use Android devices designed for use with a TV as quasi-desktops. And, starting in 2016, we have Android available on some Chrome OS devices, most of which rely on keyboard and mouse/trackpad input.

Over time, more and more Android users are going to be expecting Android apps to behave like desktop apps with respect to keyboards and mice. Some of this capability will be built into Android. Some of this capability will need to be handled by apps or libraries.

In this chapter, we will explore various techniques for making your Android app more friendly to keyboards and mice.

Prerequisites

Understanding this chapter requires that you have read the core chapters. Many of the examples use `RecyclerView`, so you may wish to review that chapter if you have not used `RecyclerView` very much. Also, some of the examples are based on drag-and-drop samples covered elsewhere in the book.

Offering Keyboard Shortcuts

One thing that users will expect from desktop-style apps is the ability to use keyboard shortcuts. Basic keyboard navigation comes "for free" for a lot of Android use cases, though in some situations you will need to add in your own keyboard

smarts, such as for navigating a `RecyclerView`. And some keyboard shortcuts will come automatically, such as Ctrl-C and Ctrl-V for copy and paste within an `EditText`.

Anything beyond that, though, you would have to provide yourself.

Action Bar Item Shortcuts

The simplest way to add keyboard shortcuts is to use `android:alphabeticShortcut` or `android:numericShortcut` on your `<item>` elements in a `<menu>` resource that you use to populate an action bar. Android will automatically support those in concert with the Ctrl key. So, for example, if you had:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/play_video"
    android:alphabeticShortcut="p"
    android:icon="@drawable/ic_movie_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_video" />
  <item
    android:id="@+id/show_thumbnail"
    android:alphabeticShortcut="t"
    android:icon="@drawable/ic_insert_photo_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_thumbnail" />
</menu>
```

(from KBMouse/HotkeysN/app/src/main/res/menu/actions.xml)

then your `onOptionsItemSelected()` method would be called not only if the user taps on the action bar items on-screen, but also if the user pressed Ctrl-P or Ctrl-T on the keyboard.

As the names suggest, `android:alphabeticShortcut` takes a letter and `android:numericShortcut` takes a number. However, try to avoid overriding existing shortcuts with unrelated logic. For example, you might consider using `android:alphabeticShortcut="v"` for a “play video” action, but that would conflict with the Ctrl-V shortcut used for paste. You would be better off going with `android:alphabeticShortcut="p"` to avoid the conflict.

In Android 8.0+, in Java, we can call variations on `setAlphabeticShortcut()`, `setNumericShortcut()`, and `setShortcuts()` that allow us to change the modifier keys from the default. There are equivalent `android:alphabeticModifiers` and `android:numericModifiers` attributes for `<item>` elements in a menu resource, to indicate the modifier keys to be used with alphabetic and numeric shortcuts.

Arbitrary Hotkeys

You may find that the action bar approach is insufficient:

- It may not make sense to have action bar items for the particular operations that you want to offer keyboard shortcuts for
- You may want to support key combinations other than Ctrl and a letter or number

You are welcome to make anything be a keyboard shortcut or "hotkey", by overriding the appropriate `KeyEvent` methods in an `Activity` or `View`.

The `KBMouse/Hotkeys` sample project is a clone of the `DragDrop/Simple` sample app from the chapter on supporting drag-and-drop. As with the original app, we show a list of available videos on the device, which the user can play or view a larger thumbnail from the video. However, in addition to drag-and-drop as a way of doing those things, this sample app also supports keyboard shortcuts:

- Alt-Right to play the video
- Ctrl-Right to view the large thumbnail

However, these keyboard shortcuts imply that the user has chosen a video to play. So, this sample app blends in the keyboard-enabled `RecyclerView` code from the corresponding section in the `RecyclerView` chapter. So, as the user presses the Down and Up arrow keys, the chosen row is highlighted. That will be the video that we work with, if the user then goes and presses Alt-Right or Ctrl-Right.

This means that we need our play-the-video and show-the-large-thumbnail code to be accessible from multiple entry points, so we pull those out into dedicated `playVideo()` and `showLargeThumbnail()` methods that take the video `Uri` as input:

```
  private void playVideo(Uri videoUri) {
    player.setVideoURI(videoUri);
```

```
    player.start();
  }

  private void showLargeThumbnail(Uri videoUri) {
    Picasso.with(thumbnailLarge.getContext())
      .load(videoUri.toString())
      .fit().centerCrop()
      .placeholder(R.drawable.ic_media_video_poster)
      .into(thumbnailLarge);
  }
```

(from KBMouse/Hotkeys/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

We then use those from the `ACTION_DROP` processing, for when the user drops the video into either the `VideoView` (referenced in the `player` field) or the `ImageView`:

```
      case DragEvent.ACTION_DROP:
        ClipData.Item clip=event.getClipData().getItemAt(0);
        Uri videoUri=clip.getUri();

        if (v==player) {
          playVideo(videoUri);
        }
        else {
          showLargeThumbnail(videoUri);
        }

        break;
```

(from KBMouse/Hotkeys/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

Handling the keyboard shortcuts is relatively straightforward, courtesy of the `onKeyDown()` callback that we override:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode==KeyEvent.KEYCODE_DPAD_RIGHT && event.getRepeatCount()==0) {
    int position=adapter.getCheckedPosition();

    if (position>=0) {
      Uri videoUri=adapter.getVideoUri(position);

      if (event.isAltPressed()) {
        playVideo(videoUri);
      }
      else if (event.isCtrlPressed()) {
        showLargeThumbnail(videoUri);
```

```
      }

      return(true);
    }
  }

  return(super.onKeyDown(keyCode, event));
}
```

(from KBMouse/Hotkeys/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

We are passed an `int` keycode (`keyCode`) and the full `KeyEvent` for whatever key that the user pressed. If the main key was Right (identified as `KEYCODE_DPAD_RIGHT` for historical reasons, and to support D-pad directional navigation options), we find out which row in the `RecyclerView` is checked, if any. If we have a checked row, we find out what the `Uri` is of the video, then call `isAltPressed()` and `isCtrlPressed()` on the `KeyEvent` to find out which modifier key was pressed in conjunction with Right, if any. If we have a match, we call the associated `playVideo()` or `showLargeThumbnail()` method.

`onKeyDown()` tends to model user expectations, in that the user expects the event to occur when the key is pressed. However, if the user continues holding down the key, we will get a stream of `onKeyDown()` calls. That is why we also check `getRepeatCount()`, to ignore the repeated keypresses, so we only try playing the video or showing the large thumbnail once if the user holds down Alt-Right or Ctrl-Right.

Android 7.0 Keyboard Shortcuts Helper

The next challenge is letting the user know what keyboard shortcuts are available. Historically, our primary option would be to hope that the user reads the app's documentation.

(you can stop laughing now)

Android 7.0 recognizes this and provides a system-wide keyboard shortcuts helper. The user can invoke this using one keyboard shortcut that (hopefully) the user will remember: Meta-/. On a Windows-centric keyboard, the Meta key is the one with the Windows logo on it.

On pre-N devices, you could offer your own keyboard shortcut mapped to Alt-/ and pop up your own keyboard shortcut dialog. Since `/` is neither a letter or a

number, and since having a keyboard shortcut action bar item might not make sense, you would do this using the `onKeyDown()` technique profiled in the previous section.

In the KBMouse/HotkeysN sample project, we will see how to:

- Add our own information to the keyboard shortcuts helper
- Show our own keyboard shortcuts helper on pre-N devices
- Use action bar item alphabetic shortcuts

Mostly, the project is a clone of the hotkey sample shown above, with the `build.gradle` file updated to build for Android 7.0:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.android.support:recyclerview-v7:24.1.1'
    implementation 'com.squareup.picasso:picasso:2.5.2'
}

android {
    compileSdkVersion 24
    buildToolsVersion '26.0.2'

    defaultConfig {
        applicationId "com.commonsware.android.kbmouse.hotkeys.n"
        minSdkVersion 23
        targetSdkVersion 24
    }
}
```

(from KBMouse/HotkeysN/app/build.gradle)

We have a menu resource now for our action bar, which happens to be the one shown towards the start of this chapter:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/play_video"
    android:alphabeticShortcut="p"
    android:icon="@drawable/ic_movie_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_video" />
  <item
    android:id="@+id/show_thumbnail"
```

```
    android:alphabeticShortcut="t"
    android:icon="@drawable/ic_insert_photo_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_thumbnail" />
</menu>
```

(from KBMouse/HotkeysN/app/src/main/res/menu/actions.xml)

We inflate that menu resource in `onCreateOptionsMenu()` using the typical recipe:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    return(super.onCreateOptionsMenu(menu));
  }
```

(from KBMouse/HotkeysN/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

In `onOptionsItemSelected()`, we need to confirm that the user has selected a row in the `RecyclerView` using the keyboard. If that is the case, we can play the video or show the thumbnail, depending upon which action bar item the user used. Otherwise, we show a `Toast` to point out the problem:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    int position=adapter.getCheckedPosition();

    if (item.getItemId()==R.id.play_video) {
      if (position>=0) {
        playVideo(adapter.getVideoUri(position));
      }
      else {
        Toast.makeText(this, R.string.msg_choose,
          Toast.LENGTH_LONG).show();
      }

      return(true);
    }
    else if (item.getItemId()==R.id.show_thumbnail) {
      if (position>=0) {
        showLargeThumbnail(adapter.getVideoUri(position));
      }
      else {
        Toast.makeText(this, R.string.msg_choose,
          Toast.LENGTH_LONG).show();
      }
```

```
    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from KBMouse/HotkeysN/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

Alternatively, you might elect to disable or hide those action bar items until the user selects a row with the keyboard.

We do not need to do anything special in our code to handle the alphabetic shortcuts — those are applied by Android automatically, routing to the same `onOptionsItemSelected()`. In other words, whether the user chooses the action bar item via a keyboard, mouse, or touchscreen, our same code runs.

The user will find out about those shortcuts through a keyboard shortcuts helper. On Android 7.0 and higher, the system will provide one for us. Note that this helper is implemented as a system-supplied dialog-themed activity. As such, our activity is paused (as we no longer get input) but not stopped (as the helper dialog is not full-screen).

We do not need to do anything special in our code to enable the keyboard shortcuts helper on Android 7.0. However, that helper only knows about our action bar item alphabetic shortcuts, plus system-wide shortcuts. It does not know anything about the Alt-Right and Ctrl-Right shortcuts that we are handling ourselves. However, we can override `onProvideKeyboardShortcuts()` in our activity to add information to this dialog about our custom shortcuts:

```
@Override
public void onProvideKeyboardShortcuts(
  List<KeyboardShortcutGroup> data, Menu menu, int deviceId) {
  super.onProvideKeyboardShortcuts(data, menu, deviceId);

  List<KeyboardShortcutInfo> shortcuts=new ArrayList<>();
  String caption=getString(R.string.menu_video);

  shortcuts.add(new KeyboardShortcutInfo(caption,
    KeyEvent.KEYCODE_DPAD_RIGHT, KeyEvent.META_ALT_ON));

  caption=getString(R.string.menu_thumbnail);
  shortcuts.add(new KeyboardShortcutInfo(caption,
    KeyEvent.KEYCODE_DPAD_RIGHT, KeyEvent.META_CTRL_ON));
```

```
    data.add(new KeyboardShortcutGroup(getString(R.string.msg_custom),
      shortcuts));
  }
```

(from KBMouse/HotkeysN/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

This is not especially well documented at this point. What seems to work is:

- Create a new `List` of `KeyboardShortcutInfo` objects
- Add one of those for each custom shortcut, via the `KeyboardShortcutInfo` constructor, where you provide a description, the primary key, and the modifier (e.g., `META_ALT_ON`) for the shortcut
- Wrap the `List` of `KeyboardShortcutInfo` objects in a `KeyboardShortcutGroup`, with your own caption for the group
- Add that `KeyboardShortcutGroup` to the passed-in `List`

This gives us:

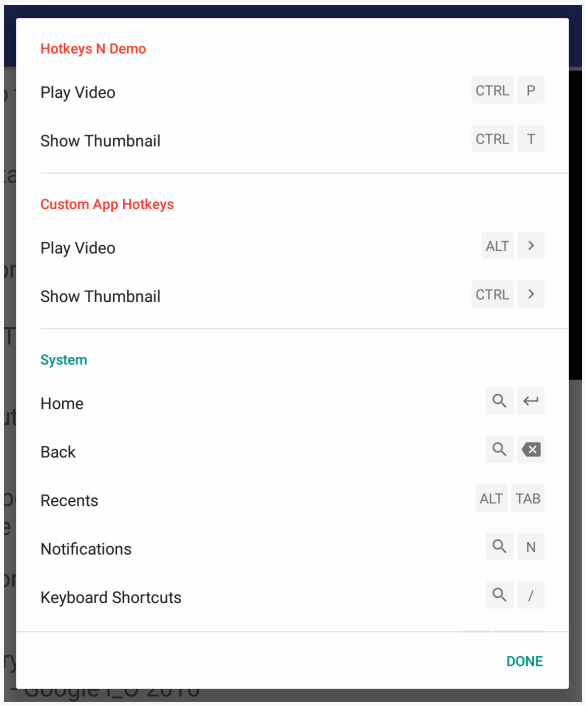


Figure 725: Android 7.0 Keyboard Shortcuts Helper, With Custom Info

The alphabetic shortcuts from the menu appear in a group whose name matches our activity's label. That is followed by our "Custom App Hotkeys" group. Ideally, these two groups would be merged, since both lists are fairly short and both pertain to this

app. While we might be able to retrieve the existing group from the supplied list of `KeyboardShortcutGroup` objects and modify it, since that is not documented, that is not safe.

This shortcut helper dialog is only available on Android 7.0. For consistency, it might be nice to offer a similar helper on older devices. To do that, we need to find out when the user presses Meta-/ on those older devices, which we handle in `onKeyDown()`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (event.getRepeatCount()==0) {
    if (keyCode==KeyEvent.KEYCODE_DPAD_RIGHT) {
      int position=adapter.getCheckedPosition();

      if (position>=0) {
        Uri videoUri=adapter.getVideoUri(position);

        if (event.isAltPressed()) {
          playVideo(videoUri);
        }
        else if (event.isCtrlPressed()) {
          showLargeThumbnail(videoUri);
        }

        return(true);
      }
    }
    else if (keyCode==KeyEvent.KEYCODE_SLASH &&
      event.isMetaPressed() &&
      Build.VERSION.SDK_INT<Build.VERSION_CODES.N) {
      new ShortcutDialogFragment().show(getFragmentManager(),
        "shortcuts");

      return(true);
    }
  }

  return(super.onKeyDown(keyCode, event));
}
```

(from KBMouse/HotkeysN/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

On those devices, we show a `ShortcutDialogFragment`, which just displays an `AlertDialog` with some simple text about our shortcuts:

```
package com.commonsware.android.kbmouse.hotkeys;

import android.annotation.TargetApi;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.os.Build;
import android.os.Bundle;

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public class ShortcutDialogFragment extends DialogFragment {
  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder=new AlertDialog.Builder(getActivity());
    Dialog dlg=builder
      .setTitle(R.string.title_shortcuts)
      .setMessage(R.string.msg_shortcuts)
      .setPositiveButton(android.R.string.ok, null)
      .create();

    return(dlg);
  }
}
```

(from KBMouse/HotkeysN/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/ShortcutDialogFragment.java)

Not every device will have a Meta key — the Pixel C's keyboard dock, for example, lacks this key. However, we have no good way of determining if a given hardware keyboard has a Meta key. Our options would be:

- To offer keyboard shortcuts help using a different key combination, even if that is non-standard with respect to Android 7.0
- Ignore the problem, not offering keyboard shortcuts help on such devices, until such time as there is an official solution

Custom Copy-and-Paste

Since the alphabetic shortcuts for action bar items are triggered via the Ctrl key, one can have action bar items for Ctrl-C and Ctrl-V, offering custom copy and paste shortcuts for your activity. This is particularly important for accessibility, when your app is reliant upon something else to get content from one point to another. For example, custom copy and paste options could be used as alternatives to drag and drop, including Android 7.0's cross-app drag-and-drop.

The KBMouse/CopyPaste sample project is a clone of the cross-app drag-and-drop sample, with two app modules: drag/ and drop/. This time, though, we are also going to allow the photo to be copied to the clipboard in drag/ and pasted from the clipboard in drop/, as a more accessible alternative. As a side benefit, this will work even on single-window environments, whereas cross-app drag-and-drop assumes that both apps are visible at the same time.

The drag/ app has a menu resource that contains our copy item, with android:alphabeticShortcut set to c:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/copy"
    android:alphabeticShortcut="c"
    android:icon="@drawable/ic_content_copy_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@android:string/copy" />
</menu>
```

(from KBMouse/CopyPaste/drag/src/main/res/menu/actions.xml)

We load that resource in onCreateOptionsMenu() as normal. In onOptionsItemSelected(), we implement the copy by getting the ClipboardManager system service and calling setPrimaryClip() to populate the clipboard:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.copy) {
      getSystemService(ClipboardManager.class)
        .setPrimaryClip(buildClip());
      Toast
        .makeText(this, R.string.msg_copy, Toast.LENGTH_SHORT)
        .show();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from KBMouse/CopyPaste/drag/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

Note that we use the API Level 23+ edition of `getSystemService()`, the one that takes the Java class object for the system service as a parameter (instead of a service name) and returns to us the system service already in the proper data type.

What we put on the clipboard is the result of `buildClip()`, which creates a `ClipData` around a `Uri` to the photo that we are publishing from this app:

```
  private ClipData buildClip() {
    Uri uri=PROVIDER
      .buildUpon()
      .appendEncodedPath(StreamProvider.getUriPrefix(AUTHORITY))
      .appendEncodedPath("assets/FreedomTower-Morning.jpg")
      .build();

    return(ClipData.newRawUri(getString(R.string.msg_photo), uri));
  }
```

(from KBMouse/CopyPaste/drag/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

We happen to be using `StreamProvider` to serve the photo straight from assets, and so we are building up a `Uri` pointing to that asset.

The `drop/` app also has a menu resource, this one set up for a `paste` action tied to Ctrl-V:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/paste"
    android:alphabeticShortcut="v"
    android:icon="@drawable/ic_content_paste_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@android:string/paste" />
</menu>
```

(from KBMouse/CopyPaste/drop/src/main/res/menu/actions.xml)

And here, `onOptionsItemSelected()` handles the `paste` request by getting the `Uri` off the clipboard and using that, in much the same way that we use the `Uri` dropped on our UI:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.paste) {
      boolean handled=false;
```

```
    ClipData clip=
      getSystemService(ClipboardManager.class)
        .getPrimaryClip();

    if (clip!=null) {
      ClipData.Item clipItem=clip.getItemAt(0);

      if (clipItem!=null) {
        imageUri=clipItem.getUri();

        if (imageUri!=null) {
          showThumbnail();
          handled=true;
        }
      }
    }

    if (!handled) {
      Toast
        .makeText(this, "Could not paste an image!", Toast.LENGTH_LONG)
        .show();
    }

    return(handled);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from KBMouse/CopyPaste/drop/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

Note that we do not do any MIME type checking to see if the `Uri` points to an image that we can use. We are relying on Picasso to show an error image if it is unable to use the `Uri`, as part of the `showThumbnail()` processing:

```
private void showThumbnail() {
  Picasso.with(this)
    .load(imageUri)
    .fit().centerCrop()
    .placeholder(R.drawable.ic_photo_size_select_actual_black_24dp)
    .error(R.drawable.ic_error_black_24dp)
    .into(image);
}
```

(from KBMouse/CopyPaste/drop/src/main/java/com/commonsware/android/dragdrop/MainActivity.java)

A more sophisticated app might check to see if the MIME type associated with the `Uri` made sense, using either `ContentResolver` or `DocumentFile`. That way, we could offer a custom error message if the user attempted to copy something else.

However, we *do* have crude logic to handle:

- an empty clipboard
- a `ClipData` with no items
- a `ClipData` without a `Uri` (e.g., plain text copied to the clipboard)

Physical Keyboards and Focusing

In an article complaining about Android's tablet support, Ars Technica's Ron Amodeo pointed out an interesting problem for devices with physical keyboards: setting the focus properly when an activity starts up. This can be addressed in a number of ways; this section shows how you can use a custom `BindingAdapter` and the data binding framework to help manage this problem.

The Problem

Suppose that you have a form that has a mix of widgets in it, including one or more `EditText` widgets. In particular, suppose that one of the `EditText` widgets is near the beginning of the activity's layout.

If you allow that widget to get the focus when the activity starts up, and the device does not have a physical keyboard, the input method editor (IME, a.k.a., "soft keyboard") appears immediately. This can be annoying to users, as perhaps they do not want to type anything into that `EditText`. Depending on the orientation of the phone and other settings, either the IME ties up a bunch of screen space that could be used for other things, or the user cannot see anything other than the full-screen landscape IME.

To save the user having to keep dismissing the IME, you might elect to give some other widget the focus at the outset.

However, for devices *with* a physical keyboard, no IME will appear. And, for users who *do* want to start typing right away, having the focus lie elsewhere is aggravating.

It would be cool if Android offered a `android:requestFocus` attribute that took a few possible values:

- `true`, meaning this widget always got the focus (replacing the `<requestFocus/>` child element used for this today)
- `false`, the default
- `ifHardKeyboard`, meaning this widget should get the focus for devices with a physical keyboard
- `ifNoHardKeyboard`, meaning this widget should get the focus for devices without a physical keyboard

Layouts might then use `android:requestFocus` on two widgets, one with `ifHardKeyboard` and one with `ifNoHardKeyboard`, to designate which widget should get the focus in either case.

Alas, this attribute does not exist. However, with a bit of work, and the assistance of the data binding framework, we can implement something fairly close to it, as is demonstrated in the `DataBinding/Focus` sample project.

A requestFocus BindingAdapter

To invent new `View` attributes, like the proposed `requestFocus` attribute, we need a `BindingAdapter`. The `@BindingAdapter` annotation can be applied to a `static` method that takes a `View` (or some subclass, if it only applies to certain view types) and a parameter indicating what sort of attribute value it expects. Then, it is up to the method to actually update the `View` as needed.

So, the `RequestFocusActivity` that demonstrates this feature has a `bindRequestFocus()` `@BindingAdapter` method:

```
@BindingAdapter("app:requestFocus")
public static void bindRequestFocus(View v, String focusMode) {
  Configuration cfg=v.getResources().getConfiguration();
  boolean hasNoKeyboard=
    cfg.keyboard==Configuration.KEYBOARD_NOKEYS;
  boolean keyboardHidden=
    cfg.hardKeyboardHidden==Configuration.HARDKEYBOARDHIDDEN_YES;
  boolean result=false;

  if (TRUE.equals(focusMode)) {
    result=true;
  }
  else if (IF_HARD_KEYBOARD.equals(focusMode)) {
    if (!hasNoKeyboard && !keyboardHidden) {
      result=true;
    }
  }
  else if (IF_NO_HARD_KEYBOARD.equals(focusMode)) {
    if (hasNoKeyboard || keyboardHidden) {
      result=true;
```

```
      if (hasNoKeyboard) v.setFocusableInTouchMode(true);
    }
  }
  else {
    throw new IllegalArgumentException("Unexpected focusMode value: "+focusMode);
  }

  if (result) {
    v.setFocusable(true);
    v.requestFocus();
  }
}
```

(from DataBinding/Focus/app/src/main/java/com/commonsware/android/focusbinding/RequestFocusActivity.java)

Unfortunately, there is no obvious way to constrain a `BindingAdapter` to some custom enumerated roster of values. So, we have `bindRequestFocus()` accept a `String`, and we throw an `IllegalArgumentException` if the attribute value is not `true`, if the attribute is not one of three recognized values:

```
private static final String TRUE="true";
private static final String IF_HARD_KEYBOARD="ifHardKeyboard";
private static final String IF_NO_HARD_KEYBOARD="ifNoHardKeyboard";
```

(from DataBinding/Focus/app/src/main/java/com/commonsware/android/focusbinding/RequestFocusActivity.java)

(`false` is skipped as a potential value, but you could easily extend this to accept and ignore that value)

The key portion of this method is determining whether or not the user has access to a physical keyboard. This has two components:

1. Does the device have a physical keyboard?
2. Is the physical keyboard presently available?

The latter might be "no" for devices with a sliding keyboard, where the keyboard exists but is not presently exposed.

Using a `Configuration` object, we set up `hasNoKeyboard` and `keyboardHidden` `boolean` values for those two components. Then, we will call `setFocusable(true)` and `requestFocus()` on the `View` if:

- the attribute value is `true`, or
- the attribute value is `ifHardKeyboard` *and* the device has a physical keyboard *and* the keyboard is not hidden, or
- the attribute value is `ifNoHardKeyboard` *and* either the device lacks a physical keyboard or the physical keyboard is hidden

We also call `setFocusableInTouchMode(true)` if the device has no physical keyboard, to ensure that the widget in question can be focused in touch mode before we try setting its focus.

Using the BindingAdapter

The syntax for our attributes is a bit different than what we aimed for:

- We use the `app` prefix rather than `android`
- The value has to be a binding expression, otherwise the data binding framework ignores it, so we cannot just use a simple string, but instead have to wrap it in an expression (e.g., `@{"ifNoHardKeyboard"}`)

But, we can use `app:requestFocus` as needed, such as in this layout:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
      android:id="@+id/button1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/a_button"
      app:requestFocus='@{"ifNoHardKeyboard"}' />

    <EditText
      android:id="@+id/editText1"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:hint="@string/str_1st_field"
      android:inputType="text"
      app:requestFocus='@{"ifHardKeyboard"}' />

    <EditText
      android:id="@+id/editText2"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:hint="@string/str_2nd_field"
```

```
      android:inputType="text" />

  </LinearLayout>
</layout>
```

(from DataBinding/Focus/app/src/main/res/layout/request_focus.xml)

Here, if the device has an available physical keyboard, we put the focus on the first `EditText` widget, so that the user can start typing right away. But, if the device lacks an available physical keyboard, we put the focus on the `Button`, to avoid the IME appearing right away.

Offering Mouse Context Menus

One thing that users of mice and trackpads are used to having are context menus, typically displayed as the result of a right-mouse click. More specifically, they are used to a popup menu appearing adjacent to the mouse pointer when they click the right-mouse button.

Android has had its own context menu system since API Level 1. However, it does not look a lot like what desktop users are used to. We can create such a menu ourselves, but it takes a little work, due to bugs and limitations in Android.

Ideally, we would use `PopupMenu`. This does pretty much what the class name implies: displays a popup window containing a menu, driven by a menu resource. However, that popup window will appear to drop down from some anchor `View` that we specify, and there is no way in the public API to adjust its position to be closer to the mouse pointer. Hence, for larger widgets — such as rows in a `ListView` or `RecyclerView` — `PopupMenu` will result in a menu that can appear fairly far away from the mouse pointer, which will be aggravating.

`PopupWindow` and `ListPopupWindow` both allow for fine-grained positioning, which make them better candidates for our purposes. Of the two, `ListPopupWindow` handles the notion of a scrolling list, which may be useful for longer menus. And, we can populate its contents from a simple `ListAdapter`, like an `ArrayAdapter`. The `KBMouse/Context` sample project is a clone of the `KBMouse/Hotkeys` sample app that adds in a `ListPopupWindow` for a mouse-driven context menu... but it takes a bit of work.

First, we need to define what goes in the list of the `ListPopupWindow`. A simple solution for that is to use a `string-array` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="popup">
    <item>@string/menu_video</item>
    <item>@string/menu_thumbnail</item>
  </string-array>
</resources>
```

(from KBMouse/Context/app/src/main/res/values/arrays.xml)

Here, the two items are references to string resources, for internationalization purposes (in theory, at least).

All of our business logic for adding the context menu lies in the `RowController` — just as it handles clicks (to play the video in a standalone player) and long-clicks (to initiate a drag-and-drop), it can now handle context menus.

First, in the `RowController` constructor, we register the `RowController` as handling touch events for the row, via `setOnTouchListener()`:

```
  RowController(View row, ChoiceCapableAdapter<?> adapter) {
    super(row);
    this.adapter=adapter;

    title=(TextView)row.findViewById(android.R.id.text1);
    thumbnail=(ImageView)row.findViewById(R.id.thumbnail);

    row.setOnClickListener(this);
    row.setOnLongClickListener(this);
    row.setOnTouchListener(this);
  }
```

(from KBMouse/Context/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/RowController.java)

That requires `RowController` to implement the `View.OnTouchListener` interface, and therefore requires `RowController` to override `onTouch()`:

```
  @Override
  public boolean onTouch(View v, MotionEvent event) {
    if ((event.getButtonState() & MotionEvent.BUTTON_SECONDARY)!=0 &&
      event.getAction()==MotionEvent.ACTION_DOWN) {
      adapter.onChecked(position, true, true);

      String[] items=
        itemView
          .getContext()
```

```
        .getResources()
        .getStringArray(R.array.popup);
    ArrayAdapter<String> adapter=
      new ArrayAdapter<>(itemView.getContext(),
        android.R.layout.simple_list_item_1,
        items);
    final ListPopupWindow popup=
      new ListPopupWindow(itemView.getContext());

    popup.setAnchorView(itemView);
    popup.setHorizontalOffset((int)event.getX());
    popup.setVerticalOffset((int)event.getY()-itemView.getHeight());
    popup.setAdapter(adapter);
    popup.setWidth(measureContentWidth(itemView.getContext(), adapter));

    popup.setOnItemClickListener(
      new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view,
                                int position, long id) {
          if (position==0) {
            ((MainActivity)itemView.getContext())
              .playVideo(videoUri);
          }
          else {
            ((MainActivity)itemView.getContext())
              .showLargeThumbnail(videoUri);
          }

          popup.dismiss();
        }
      });

    popup.show();

    return(true);
  }

  return(false);
}
```

(from KBMouse/Context/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/RowController.java)

To determine if a given `MotionEvent` is a trigger for the context menu, we check two things:

1. Is the `BUTTON_SECONDARY` pressed?

2. Is this a "down" event (`ACTION_DOWN`)?

If yes, we tell our `ChoiceCapableAdapter` to check this row, so it is clear to the user what they have right-clicked over.

Then, we load in the string array from the resources and wrap that in a standard `ArrayAdapter`. At that point, we can begin configuring the `ListPopupWindow`.

The constructor for the `ListPopupWindow` takes a `Context`. We grab the `Context` from the `itemView` field of this `ViewHolder`, which represents our row.

We then call five configuration methods to set up the look-and-feel of the `ListPopupWindow`. The first four are fairly straightforward:

- `setAnchorView()` specifies the `View` that this popup is anchored to. In this case, we use the row itself (`itemView`).
- `setHorizontalOffset()` and `setVerticalOffset()` indicate, from the upper-left corner of the anchor view, where to place the upper-left corner of the `ListPopupView`. We want the `ListPopupView` to be adjacent to the mouse pointer, and the `getX()` and `getY()` values of the `MotionEvent` tell us where inside the `itemView` the user clicked. However, the *default* position of the `ListPopupWindow` is to be anchored to the *lower left corner* of the anchor view, not the upper left corner. To adjust the horizontal position, we can simply use `getX()`, since both `getX()` and the default horizontal position of the `ListPopupWindow` are on the left. However, the offset for the vertical position needs to be a negative value, as we want to raise the `ListPopupWindow` to where the mouse pointer is. That value is the difference between the Y coordinate of the mouse pointer (`getY()`) and the height of the row (`itemView.getHeight()`).
- `setAdapter()` provides the `ArrayAdapter` to populate the list in the `ListPopupWindow`.

The fifth method — `setWidth()` — is more complex than it should be, due to a bug. Ideally, we would call `setWidth(ListPopupWindow.WRAP_CONTENT)`. According to the documentation, this will set the width of the `ListPopupWindow` to be the width of the content of the adapter. Unfortunately, this does not work. And, since that bug has been outstanding since 2013, it is unlikely that it will ever work.

The workaround — as documented in Stack Overflow – is to calculate the maximum width of our adapter rows ourselves, then call `setWidth()` with that pixel value. This is isolated in a `measureContentWidth()` method:

```
// based on http://stackoverflow.com/a/26814964/115145

private int measureContentWidth(Context ctxt, ListAdapter listAdapter) {
  ViewGroup mMeasureParent = null;
  int maxWidth = 0;
  View itemView = null;
  int itemType = 0;

  final ListAdapter adapter = listAdapter;
  final int widthMeasureSpec =
    View.MeasureSpec.makeMeasureSpec(0, View.MeasureSpec.UNSPECIFIED);
  final int heightMeasureSpec =
    View.MeasureSpec.makeMeasureSpec(0, View.MeasureSpec.UNSPECIFIED);
  final int count = adapter.getCount();
  for (int i = 0; i < count; i++) {
    final int positionType = adapter.getItemViewType(i);
    if (positionType != itemType) {
      itemType = positionType;
      itemView = null;
    }

    if (mMeasureParent == null) {
      mMeasureParent = new FrameLayout(ctxt);
    }

    itemView = adapter.getView(i, itemView, mMeasureParent);
    itemView.measure(widthMeasureSpec, heightMeasureSpec);

    final int itemWidth = itemView.getMeasuredWidth();

    if (itemWidth > maxWidth) {
      maxWidth = itemWidth;
    }
  }

  return maxWidth;
}
```

(from KBMouse/Context/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/RowController.java)

This implementation iterates over the items in the adapter, has the adapter create row views for each row (using a bit of light caching to try to recycle row views in simple adapters), then determines the measured width of those rows. The longest measured width is then used as the result. This works for simple rows and short lists, which is all we really need here anyway.

Then, we call `setOnItemClickListener()` to register a listener to find out when rows in the list in the `ListPopupWindow` are clicked. This works the same as with a `ListView`. In our case, we look at the passed-in `position`, and route to the activity's `playVideo()` or `showLargeThumbnail()` methods according to which list item the user clicked upon. We then `dismiss()` the `ListPopupWindow`, so it goes away after the user clicked on one of the list items.

Finally, we can `show()` the `ListPopupWindow`, so it displays its list to the user.

From the user's perspective, right-clicking over one of the videos in the list offers the context menu:

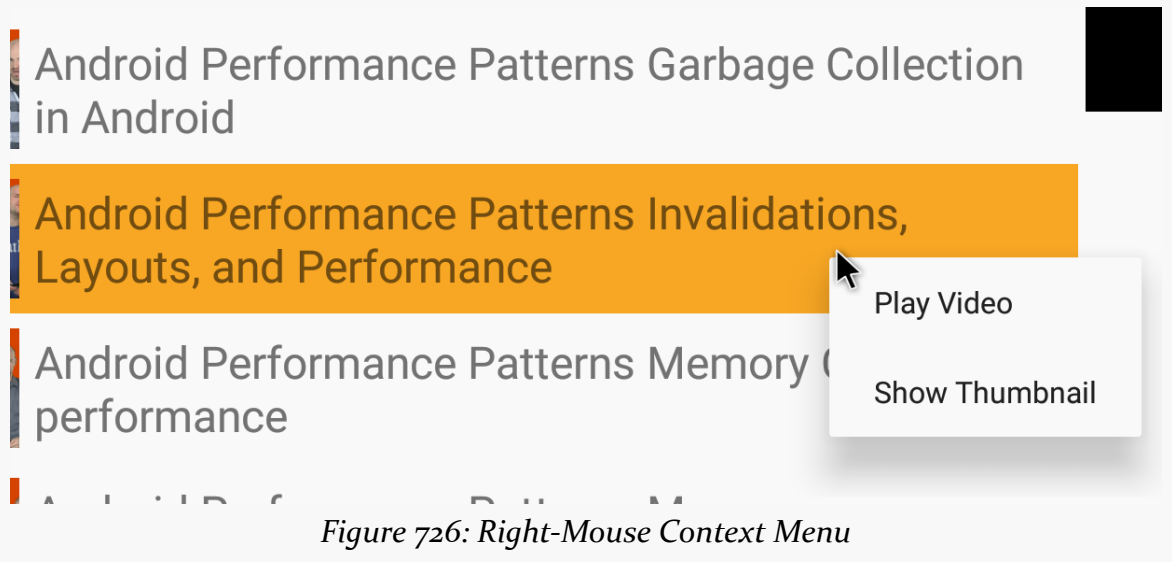


Figure 726: Right-Mouse Context Menu

What is missing is the ability for the user to dismiss the context menu. If the user clicks outside the `ListPopupWindow`, it goes away as expected. However, the underlying click event is still processed. So, if the user clicks over the `VideoView` or large thumbnail `ImageView`, everything looks fine. If the user clicks over one of the `RecyclerView` rows... the clicked-upon video starts playing back in a standalone video player, rather than just dismissing the `ListPopupWindow`. This will be addressed in a future edition of this sample app.

Also note that API Level 23 offers `setOnContextClickListener()` on `View`. This works like `setOnClickListener()` and `setOnLongClickListener()`, letting you know via an `OnContextClickListener` that the view was "context-clicked". However, you do not get details of the touch event, and so you have no good means of positioning the popup to be near where it should go.

Offering Tooltips

Users of desktop operating systems, and even Web apps, are used to having tooltips available on key UI elements (e.g., buttons). In a mouse-driven environment, these usually appear after the mouse has hovered over the UI element for a short while.

Touchscreens can also offer similar contextual help. For example, if the user long-presses on an action bar item's icon, a `Toast` will appear with the item's title.

So, there are two pieces to the puzzle of offering tooltips:

1. How do you decide when to show a tooltip?
2. What is the UI for the tooltip itself?

As with many things in Android, there are two options, depending on what version of Android you are running on.

Android 7.1 and Older

This section will focus on the first piece — how you decide when to show a toolip — using a simple `Toast` for the tooltip UI. There are many Android libraries for tooltip UIs, which you may wish to investigate.

Hover Events

API Level 14 added `setOnHoverListener()` to `View`. This, and the corresponding `OnHoverListener` interface, allow developers to find out when the user is "hovering" over a widget. This can be triggered by a mouse or by some styluses.

The `onHover()` method of your `OnHoverListener` will be called when a change in the state of hovering occurs. You are given the affected `View`, along with the `MotionEvent` that triggered the state change. The `MotionEvent` should have one of three actions:

- `ACTION_HOVER_ENTER`, meaning that the user has begun hovering over the view
- `ACTION_HOVER_MOVE`, meaning that the user was already hovering over the view, but has moved the mouse pointer, so the hover position within the view bounds has changed
- `ACTION_HOVER_EXIT`, meaning that the user is no longer hovering over the view

The KBMouse/Tooltip sample project is a clone of the KBMouse/Hotkeys sample app that adds in tooltip support for long-clicks and hovers. MainActivity now implements the OnLongClickListener and OnHoverListener interfaces, so we can register those listeners on our VideoView and large thumbnail ImageView:

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);
  setContentView(R.layout.main);

  player=(VideoView)findViewById(R.id.player);

  if (player!=null) {
    player.setOnDragListener(this);
    player.setOnHoverListener(this);
    player.setOnLongClickListener(this);
  }

  thumbnailLarge=(ImageView)findViewById(R.id.thumbnail_large);

  if (thumbnailLarge!=null) {
    thumbnailLarge.setOnDragListener(this);
    thumbnailLarge.setOnHoverListener(this);
    thumbnailLarge.setOnLongClickListener(this);
  }

  setLayoutManager(new LinearLayoutManager(this));
  adapter=new VideoAdapter(getRecyclerView());
  setAdapter(adapter);
  getRecyclerView().requestFocus();

  if (state!=null) {
    isInPermission=
      state.getBoolean(STATE_IN_PERMISSION, false);
  }

  if (hasFilesPermission()) {
    loadVideos();
  }
  else if (!isInPermission) {
    isInPermission=true;

    ActivityCompat.requestPermissions(this,
      new String[] {Manifest.permission.READ_EXTERNAL_STORAGE},
      REQUEST_PERMS);
  }
}
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

We will see why we are not registering for events on the `RecyclerView` a bit later in this chapter.

The `onLongClick()` method for the `OnLongClickListener` forwards the event to a `showTooltip()` method, providing a string resource with the particular message to show:

```
@Override
public boolean onLongClick(View v) {
  if (v==thumbnailLarge) {
    showTooltip(R.string.tooltip_thumbnail);
  }
  else if (v==player) {
    showTooltip(R.string.tooltip_player);
  }
  else {
    return(false);
  }

  return(true);
}
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

`showTooltip()` shows a `Toast`... with a slight wrinkle:

```
private void showTooltip(@StringRes int message) {
  if (tooltip!=null) {
    tooltip.cancel();
  }

  tooltip=Toast.makeText(this, message, Toast.LENGTH_LONG);
  tooltip.show();
}
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

If, while a tooltip `Toast` is showing, the user long-clicks on a *different* widget, we want to show that tooltip immediately. By default, we would have to wait until the first `Toast` completed its duration. `Toast` has a `cancel()` method that does one of three things:

1. If the `Toast` has not yet been shown, it removes the `Toast` from the roster of pending `Toasts`

2. If the `Toast` is presently showing, it gets rid of the `Toast`
3. If the `Toast` was shown previously, it does nothing, as nothing is needed

So, we hold onto the most-recent tooltip `Toast` in a field named `tooltip`, and we `cancel()` it before showing the next tooltip.

The `onHover()` method for the `OnHoverListener` forwards the event to a private `onHover()`, where we supply the string resource of the tooltip message:

```
@Override
public boolean onHover(View v, MotionEvent event) {
  if (v==player) {
    onHover(event, R.string.tooltip_player, player);
  }
  else if (v==thumbnailLarge) {
    onHover(event, R.string.tooltip_thumbnail, thumbnailLarge);
  }
  else {
    return(false);
  }

  return(true);
}
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

And that is where things start to get complicated.

Detecting a Long-Enough Hover

The problem is that while we can find out when the user starts and stops hovering over a view, we need additional code to determine when some time has elapsed between those two events. So, for example, if you want to show a tooltip one second after the user begins hovering over a view, you need to:

- Arrange to find out when one second has elapsed after the user starts hovering over the view
- Cancel that work if the user stops hovering over that view, either preventing the tooltip from appearing (if it has not done so yet) or perhaps causing the tooltip to vanish immediately (if it is presently being displayed)

`postDelayed()` provides a nice cheap way of handling this. We can schedule a `Runnable` to be invoked after our one-second delay, and we can use

`removeCallbacks()` to cancel the `Runnable` if the user stops hovering over that widget.

However, we do not know the exact sequence of hover events. It may be that we are given an `ACTION_HOVER_ENTER` event for a new widget prior to `ACTION_HOVER_EXIT` of the previous widget. Hence, we need to track per-widget state.

In this case, the sample app holds onto a `Runnable` per widget for which we are displaying tooltips, in a `SparseArray` object:

```
private SparseArray<Runnable> hoverTimers=
  new SparseArray<>();
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

A `SparseArray` is a data structure that allows `ArrayList`-style indexed access to objects, but it does not assume that the index values are consecutive. Basically, it replaces a `HashMap` of `Integer` objects. The `SparseArray` allows us to use widget IDs as the index values, so as long as each of our tooltip-enabled widgets have unique IDs, we can track their state in the `SparseArray`.

Our private edition of `onHover()` uses this `SparseArray`:

```
private void onHover(MotionEvent event, @StringRes final int message,
                     final View anchor) {
  Runnable hover=hoverTimers.get(anchor.getId());

  if (hover==null &&
    (event.getAction()==MotionEvent.ACTION_HOVER_ENTER ||
    event.getAction()==MotionEvent.ACTION_HOVER_MOVE)) {

    hover=new Runnable() {
      @Override
      public void run() {
        showTooltip(message);
      }
    };

    hoverTimers.put(anchor.getId(), hover);
    thumbnailLarge.postDelayed(hover, TOOLTIP_DELAY);
  }
  else if (hover!=null &&
    event.getAction()==MotionEvent.ACTION_HOVER_EXIT) {
    thumbnailLarge.removeCallbacks(hover);
    hoverTimers.remove(anchor.getId());
```

```
    }
  }
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

There are two scenarios that we care about:

1. The user starts hovering over a widget, and we have not yet set up the timer `Runnable`, as it does not exist in the `SparseArray`. So, we create the `Runnable` to show our tooltip (via `showTooltip()`), stuff it in the `SparseArray`, and use `postDelayed()` to trigger the `Runnable` after a second.
2. The user stops hovering over a widget, and we already have the timer `Runnable`. In this case, we call `removeCallbacks()` to ensure that this `Runnable` is not run (if it has not run already), plus remove it from the `SparseArray`.

The net effect is that our `Runnable` will be invoked if one second elapses after the user starts hovering over a widget and the user has not stopped hovering over that same widget.

`showTooltip()` raises a `Toast` to serve as a crude tooltip:

```
  private void showTooltip(@StringRes int message) {
    if (tooltip!=null) {
      tooltip.cancel();
    }

    tooltip=Toast.makeText(this, message, Toast.LENGTH_LONG);
    tooltip.show();
  }
```

(from KBMouse/Tooltip/app/src/main/java/com/commonsware/android/kbmouse/hotkeys/MainActivity.java)

However, it is possible that the user displays one tooltip, then quickly hovers over another widget, and that our one-second delay is shorter than the `Toast` display duration. So, we hold onto each `Toast` that we display, and we `cancel()` it before showing the next one. That way, if one `Toast` is outstanding when we need to show a different `Toast`, we can make the switch immediately, rather than having to wait for the rest of the first `Toast` display duration to elapse before showing the next one.

What We Are Missing

However, the tooltip for the `VideoView` will not display. `VideoView` is an odd widget, consuming a lot of touch events for no really good reason. Most likely, for a sophisticated Android app, you will want to skip `VideoView` and use something else, such as a `MediaPlayer` tied to your own `SurfaceView` or `TextureView`.

This sample app does not attempt to provide tooltips for the rows in the `RecyclerView`. Even though we are set up to show a tooltip for the `RecyclerView` overall, the hover events are for the rows, not the `RecyclerView`, as the `RecyclerView` itself is not visible. If you wanted tooltips here, your choices are:

- Have the tooltip for the whole `RecyclerView`, forwarding hover events from rows up to your tooltip-management logic
- Have tooltips per row, using some different system than our `SparseArray`, as the row widget IDs will all be the same by default

Android 8.0+

Android has had the occasional tooltip-like feature, such as the tooltip that you get on action bar items when long-pressing them. Android 8.0 extends this to all widgets. Simply provide the desired text via `android:tooltipText` or `setTooltipText()`. A tooltip will then appear when either the user long-presses the widget or if the user hovers over the widget. In both cases, though, if the long-press or hover event is consumed by something else, the tooltip is not shown.

The KBMouse/Tooltip0 sample project is a clone of the preceding sample app, where we switch to using the tooltip implementation in Android 8.0+.

Specifically, our large-screen layout has tooltips for the `VideoView` and `ImageView` widgets, using `android:tooltipText`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">

  <android.support.v7.widget.RecyclerView android:id="@+id/video_list"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="0dp"
    android:layout_height="match_parent"
```

```
    android:layout_weight="1" />

  <LinearLayout
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:orientation="vertical">

    <FrameLayout
      android:id="@+id/video_frame"
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_marginBottom="4dp"
      android:layout_weight="1"
      android:padding="4dp">

      <VideoView
        android:id="@+id/player"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center"
        android:tooltipText="@string/tooltip_player" />
    </FrameLayout>

    <FrameLayout
      android:id="@+id/thumbnail_frame"
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="1"
      android:padding="4dp">

      <ImageView
        android:id="@+id/thumbnail_large"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerInside"
        android:tooltipText="@string/tooltip_thumbnail" />
    </FrameLayout>
  </LinearLayout>
</LinearLayout>
```

(from KBMouse/TooltipO/app/src/main/res/layout-w800dp/main.xml)

The result are tooltips reminiscent of those from the original sample:

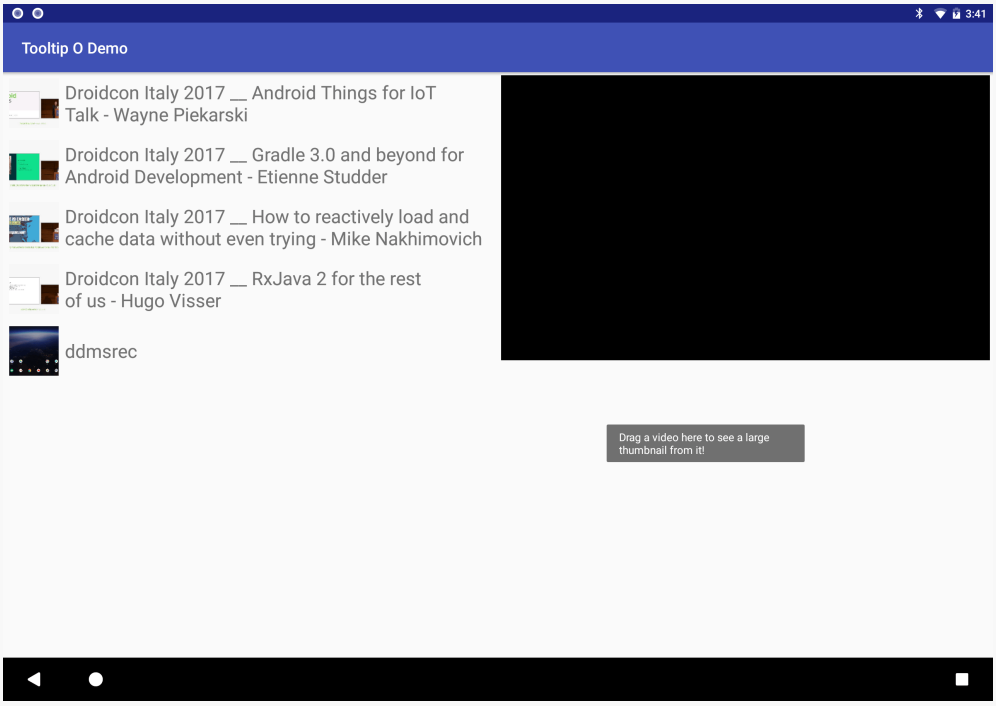


Figure 727: Tooltip From Long-Press of ImageView

Hovering over either widget, or long-pressing either widget, will bring up the tooltip.

Pointer Capture

On a desktop operating system, the "pointer" refers to the mouse cursor. This feature is normally handled by the operating system. "Pointer capture" refers to when a widget blocks the normal mouse cursor, typically because it is going to use the mouse in some way for which the normal pointer would be inappropriate.

In Android 8.0+, you can call `requestPointerCapture()` and `releasePointerCapture()` to have a widget operate in pointer-capture mode or not.

The *vast* majority of Android apps should not need this.

Viewing PDFs

The Portable Document Format — better known as PDF — has been around for over two decades, and it is still going strong today. As a result, we often have a need to show PDF files to users, whether those files are:

- Shipped with the app, such as documentation
- Downloaded by the app, such as email attachments for the mail client that you are writing
- Otherwise managed by the app, such as with PDF files held in cloud storage that your app is managing on behalf of the user
- And so on

This is another one of those topics that seems fairly simple on the surface, but can get complicated based on your requirements. In particular, if you want to try to present the PDF to the user *in your app*, as opposed to launching some external PDF viewing app, while you have a few options, they all have their issues.

In this chapter, we will review several ways of displaying a PDF to the user, so you can choose what approach (or approaches) are the best fit for your requirements.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

The Criteria

If there are several ways of displaying a PDF, we need some criteria by which to evaluate those options and make our choice. Here are some likely criteria to use, though you may have others that are more specific to your app's requirements.

Where is the PDF?

Most of the PDF-viewing options assume that the PDF is on the device already. However, even within that, there are a few possibilities as to where the PDF might be:

- Internal storage (e.g., `getFilesDir()`) of your app
- External storage (e.g., `getExternalFilesDir()`) associated with your app
- Common locations on external storage, such as the standard `Documents` directory
- Some `Uri` that you obtained via `ACTION_GET_CONTENT` or `ACTION_OPEN_DOCUMENT`
- An asset or raw resource within your app

And, occasionally, developers get boxed into a corner, with a requirement to show a PDF without downloading the PDF.

Does It Work Offline?

The opposite problem is the offline viewing experience. If the PDF is on the device already, ideally, the PDF-viewing technology can show that PDF without having to have an active Internet connection. After all, many users do not have continuous Internet access.

How Complex is the PDF?

While the PDF file format has been around for years and years, it has expanded over that time. In the beginning, PDF did not offer things like form fields, annotations, and the like. The more feature-rich your PDF is, the more likely it is that some PDF viewers may be incapable of handling all of those features.

This is not restricted to in-app viewing solutions. PDF viewing apps vary in capability, and there will be some that either do not support your desired features

(e.g., allow the user to fill in forms) or only will if the user unlocks the feature through some purchase.

How Stable is the Solution?

Many PDF-viewing apps are continuously updated. If the user happens to choose one that is not, that is the user's fault, not yours.

However, when you build in PDF viewing into your app, now *you* need to ensure that your solution is up to date and likely to stay stable for the duration of your app's use.

How Private is the PDF?

The big one, in terms of the major options for viewing the PDF, is the degree of privacy that is tied to that PDF.

If the document comes from the user — for example, it came in an email attachment to the user's inbox that you are managing — you should assume a modest level of privacy. Do not ship that PDF to other servers without the user's approval. However, if the user wants to view the PDF, you should not be afraid of allowing the user to do so using the user's chosen PDF viewing app.

If the document is more tightly controlled by your app — for example, it came as an attachment to some secure messaging client that you are writing — you may be nervous about granting other apps access to that PDF. After all, you do not necessarily know what those other apps might do with the PDF.

Where things get very messy is when the developers think that *they* "own" the PDF, and therefore are trying to restrict that PDF's access, much as how a DRM solution tries to restrict access to videos. Many PDF-viewing apps offer printing, sharing, and other things that the *user* might want, but that the *developers* do not want. In truth, it is rather likely that the user can get to the PDF anyway, whether the developers like it or not, by various means (e.g., extracting PDFs packaged in the APK using an ordinary ZIP file utility). But, this seems to be the #1 reason why some developers are trying to view PDF files within their own app: treating the user as the enemy.

The Classic Solution: ACTION_VIEW

The quintessential solution — and the one that you should focus on first — is to use `startActivity()` with an `ACTION_VIEW` `Intent`, where the `Uri` in that `Intent` points to your PDF file:

```
startActivity(new Intent(Intent.ACTION_VIEW, uriToThePdf));
```

This is quick, easy, and gives the user control over how the PDF is rendered, as they can (usually) choose the PDF viewer to use.

However, this is not a universal solution:

- Android itself does not ship with a PDF viewer app, and therefore some devices will not ship with a PDF viewer app. You wind up with an `ActivityNotFoundException` from your `startActivity()` call, and then you need to guide the user to install a PDF viewer app.
- The user may not be able to install such an app, due to limitations imposed by device owner APIs or because the user is using the device on a secondary user account that lacks app-installation capability.
- For maximum compatibility, you will need to set up a `FileProvider` (or the equivalent) and serve your PDF through it, particularly on Android 7.0+ devices, where the `file` scheme for `Uri` values is banned, in effect.

If you are concerned that the user might choose a PDF viewer app that is malware, or is one that has features that you dislike, you could use `PackageManager` and `queryIntentActivities()` to find all activities that support `ACTION_VIEW` for your PDF file. From there, you can find out the apps associated with those activities, and filter that based on a whitelist (or blacklist) of PDF viewer apps. Then, present your own "chooser"-style UI to allow the user to choose from among the valid options.

The Really Bad Idea: Google Docs

Some developers, though, try desperately to avoid using external PDF viewer apps.

The approach that many of those developers try is the worst available option: have Google Docs render the PDF in a `WebView`, using code reminiscent of:

```
WebView webview=(WebView)findViewById(R.id.webview);
```

```
webview.getSettings().setJavaScriptEnabled(true);
webview.loadUrl("http://drive.google.com/viewerng/
viewer?embedded=true&url=" + urlToYourPdf);
```

This has many poor characteristics:

- It requires that the PDF be available publicly on the Web, for Google Docs to be able to access it. From a security standpoint, many more attackers can access the PDF online than can on an Android device.
- If the PDF is not already available publicly on the Web, you would need to run a server, or employ some cloud storage or similar service, to host that PDF, adding to your complexity.
- If the PDF needs to be uploaded, now you are consuming lots of extra bandwidth. Not only did you (probably) download the PDF from somewhere, but now you have to re-upload it somewhere, then download whatever bits and bytes the `WebView` winds up using to display it, as served from Google Docs.
- There is no requirement that Google keep this Google Drive view-an-arbitrary-PDF URL working. As soon as they change it, your app breaks. As soon as they discontinue it, this approach has to be replaced anyway.
- This approach requires a live Internet connection. There is no offline option.

Pretty much anything else will be better than this solution.

The Built-In Option: PdfRenderer

Routinely, developers are astonished that Android does not ship with some sort of PDF-viewing app, the way that desktop operating systems do.

The closest thing that Android has to a built-in PDF viewer is `PdfRenderer`. This class was added in API Level 21 (Android 5.0), and it allows you to render pages of a PDF to `Bitmap` objects. In turn, you can then use those `Bitmap` objects to present the pages to the user, by one means or another.

This solution has severe limitations:

- As noted, it requires Android 5.0, leaving out older devices.
- It requires a seekable stream on the content, in the form of a `ParcelFileDescriptor`. The net effect of this is that you can only reliably render something backed by a file. Conversely, you cannot render something

backed by a pipe. This means that you cannot render assets, raw resources, a PDF that you decrypt into a memory buffer, and so on.
- It chokes on complex PDF files. For example, the author of this book started testing it with this short research paper, only to determine (eventually) that `PdfRenderer` simply cannot render that PDF. `PdfRenderer` was added with an eye towards print preview, and so its implementation may be focused on the sorts of PDFs that you can generate for printing, as opposed to arbitrary PDFs.
- It only gives you `Bitmap` objects. You still have to implement a UI to present those `Bitmap` objects to the user.

The PDF/PdfRenderer sample project illustrates the use of `PdfRenderer`.

The RecyclerView

We need to present pages to the user. In a typical PDF viewer, the user can scroll or swipe to move between pages. One could use a `ViewPager` for that role. However, large `Bitmap` objects really should be reused, and `ViewPager` does not make it easy to reuse objects. A `RecyclerView` would be a better choice.

The chapter on advanced `RecyclerView` uses demonstrates setting up a `RecyclerView` as a replacement for `ViewPager` for page-at-a-time presentations. The `PdfRenderer` sample app uses one of those techniques, involving `SnapHelper`, for presenting the pages to the user.

Hence, our UI is a `RecyclerView` with a light gray background:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView android:id="@+id/pager"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="#CCCCCCCC"
  android:clipToPadding="false" />
```

(from PDF/PdfRenderer/app/src/main/res/layout/main.xml)

In `onCreate()`, our `MainActivity` sets up that `RecyclerView` with a `LinearLayoutManager`, along with a `PagerSnapHelper` named `snapperCarr`:

```
    pager.setLayoutManager(new LinearLayoutManager(this,
      LinearLayoutManager.HORIZONTAL, false));
```

```
    snapperCarr.attachToRecyclerView(pager);
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

Eventually, we will tie in a `RecyclerView.Adapter` and `RecyclerView.ViewHolder` classes, named `PageAdapter` and `PageController`. However, there is nothing to render at the outset, as we do not have a PDF to use.

Getting the PDF

Later samples in this chapter happen to use PDF rendering technologies that can work with ordinary streams. In those cases, we can package some PDFs as assets, to provide sample behavior without the user having to rummage around and find a PDF.

Alas, that is not an option with `PdfRenderer`. So, we need to let the user find a PDF on external or removable storage.

To do that, we have an action bar item named "Open" that leads to an `open()` method on `MainActivity`:

```
    Intent i=new Intent()
      .setType("application/pdf")
      .setAction(Intent.ACTION_OPEN_DOCUMENT)
      .addCategory(Intent.CATEGORY_OPENABLE);

    startActivityForResult(i, REQUEST_OPEN);
  }
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

Here, we use the Storage Access Framework, invoking an `ACTION_OPEN_DOCUMENT` `Intent`, requesting PDF files.

We get control again in `onActivityResult()`:

```
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (resultCode==Activity.RESULT_OK) {
      pickedDocument=data.getData();
      show(pickedDocument);
    }
```

```
}
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

If the user chose a PDF, we stuff its `Uri` into a `pickedDocument` field, plus call a private `show()` method:

```
    try {
      adapter=new PageAdapter(getLayoutInflater(),
        getContentResolver().openFileDescriptor(uri, "r"));
      pager.setAdapter(adapter);
    }
    catch (java.io.IOException e) {
      Log.e("PdfRenderer", getString(R.string.toast_open), e);
      Toast.makeText(this, R.string.toast_open, Toast.LENGTH_LONG).show();
    }
  }
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

`show()` creates a `PageAdapter`, giving it a `LayoutInflater` and a `ParcelFileDescriptor` on the `Uri`, obtained by calling `openFileDescriptor()` on a `ContentResolver`. We then attach `PageAdapter` to our pager-style `RecyclerView`.

To handle configuration changes, we hold onto `pickedDocument` in the saved instance state `Bundle`:

```
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putParcelable(STATE_PICKED, pickedDocument);
  }
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

...and restore it in `onCreate()`:

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    pager=(RecyclerView)findViewById(R.id.pager);
    pager.setLayoutManager(new LinearLayoutManager(this,
      LinearLayoutManager.HORIZONTAL, false));
    snapperCarr.attachToRecyclerView(pager);
```

```
    if (savedInstanceState!=null) {
      pickedDocument=savedInstanceState.getParcelable(STATE_PICKED);

      if (pickedDocument!=null) {
        show(pickedDocument);
      }
    }
  }
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

In `onDestroy()`, we call a `close()` method on `PageAdapter` — we will see the role of `close()` shortly:

```
  protected void onDestroy() {
    if (adapter!=null) {
      adapter.close();
    }

    super.onDestroy();
  }
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/MainActivity.java)

Adding the PdfRenderer

`PageAdapter` is surprisingly short and mostly is focused on implementing the `RecyclerView.Adapter` API:

```
package com.commonsware.android.pdfrenderer;

import android.graphics.pdf.PdfRenderer;
import android.os.ParcelFileDescriptor;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.ViewGroup;
import java.io.IOException;

class PageAdapter extends RecyclerView.Adapter<PageController> {
  private final LayoutInflater inflater;
  private final PdfRenderer renderer;

  PageAdapter(LayoutInflater inflater, ParcelFileDescriptor pfd)
    throws IOException {
```

```
    this.inflater=inflater;
    renderer=new PdfRenderer(pfd);
  }

  @Override
  public PageController onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new PageController(inflater.inflate(R.layout.page, parent, false)));
  }

  @Override
  public void onBindViewHolder(PageController holder, int position) {
    PdfRenderer.Page page=renderer.openPage(position);

    holder.setPage(page);
    page.close();
  }

  @Override
  public int getItemCount() {
    return(renderer.getPageCount());
  }

  void close() {
    renderer.close();
  }
}
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/PageAdapter.java)

In the constructor, we create an instance of a `PdfRenderer`, passing in the `ParcelFileDescriptor` on our PDF file.

In `onCreateViewHolder()`, we create an instance of a `PageController`, passing it an inflated layout representing the page contents — we will examine this in greater detail shortly.

In `onBindViewHolder()`, we ask the `PdfRenderer` to give us a `PdfRenderer.Page` for the page identified by our `position`, via an `openPage()` method. We then pass that to the `PageController` via a `setPage()` method, before we `close()` the `Page`. It is the responsibility of `PageController` to ensure that it does everything that it needs to do with that page before returning from `setPage()`.

`getItemCount()` returns the number of pages in the PDF, as determined by `getPageCount()`.

Finally, `close()` calls `close()` on the `PdfRenderer`, to let it release any resources that it is holding onto, such as that `ParcelFileDescriptor`.

Showing the Pages

The real fun, of course, is in `PageController`, for rendering a page of the PDF into a `Bitmap` to be shown in a page of our pager-style `RecyclerView`.

To allow for pinch-to-zoom functionality, the `PdfRenderer` sample app uses Dave Morrissey's `SubsamplingScaleImageView`:

```
<com.davemorrissey.labs.subscaleview.SubsamplingScaleImageView
  android:id="@+id/page"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_margin="16dp" />
```

(from PDF/PdfRenderer/app/src/main/res/layout/page.xml)

This has a `16dp` margin on all four sides, causing each page of the PDF to appear to be floating over the light gray background of the `RecyclerView` itself. Android's `PdfRenderer` does not put any frame around its rendered page, leaving that up to you.

`PageController` grabs the `SubsamplingScaleImageView` in the constructor, then uses `PdfRenderer.Page` and a `Bitmap` to populate it in `setPage()`:

```
package com.commonsware.android.pdfrenderer;

import android.graphics.Bitmap;
import android.graphics.pdf.PdfRenderer;
import android.os.Environment;
import android.support.v7.widget.RecyclerView;
import android.view.View;
import android.widget.ImageView;
import com.davemorrissey.labs.subscaleview.ImageSource;
import com.davemorrissey.labs.subscaleview.SubsamplingScaleImageView;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

class PageController extends RecyclerView.ViewHolder {
  private final SubsamplingScaleImageView iv;
  private Bitmap bitmap;
```

```
  PageController(View itemView) {
    super(itemView);

    iv=(SubsamplingScaleImageView)itemView.findViewById(R.id.page);
  }

  void setPage(PdfRenderer.Page page) {
    if (bitmap==null) {
      int height=2000;
      int width=height * page.getWidth() / page.getHeight();

      bitmap=Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
    }

    bitmap.eraseColor(0xFFFFFFFF);
    page.render(bitmap, null, null, PdfRenderer.Page.RENDER_MODE_FOR_DISPLAY);
    iv.resetScaleAndCenter();
    iv.setImage(ImageSource.cachedBitmap(bitmap));
  }
}
```

(from PDF/PdfRenderer/app/src/main/java/com/commonsware/android/pdfrenderer/PageController.java)

In `setPage()`, we lazy-create a `Bitmap`, set to 2000 pixels high and whatever width would be appropriate based on the aspect ratio of the page. This is a simple implementation, suitable for a PDF where all pages have the same aspect ratio. A more sophisticated sample would use some form of object pool for `Bitmap` objects based on aspect ratio.

Note that the `Bitmap` needs to be `ARGB_8888`, exacerbating its memory usage. Attempts to use `RGB_565` — thereby cutting memory usage in half per page — fail with an error from `PdfRenderer`.

Actually rendering the page to the `Bitmap` is performed by the `render()` method on the `Page`. It takes the `Bitmap`, an optional `Rect` indicating a subset of the page to be rendered, an optional `Matrix` to be applied to transform the rendering, and a "render mode" to indicate if this is for use on a screen or (somehow) on a printed page.

However, `render()` does not do anything to the `Bitmap` other than render the page's contents. In particular, it does not clear the `Bitmap` ahead of time. Since we are reusing the `Bitmap` objects, by default, as those `Bitmap` objects get reused, they would accumulate page contents, which is not what we want. Also, `render()` does not assume any particular background color for pages, if such a color was not specified in the PDF. Instead, it just renders the "ink" on top of the `Bitmap`, with

everything else left alone. So, we use `eraseColor()` to reset the `Bitmap` to white before we call `render()`, so we clear out any previous page's content, and so we have a solid white background for our pages.

Not only are we reusing the `Bitmap`, but we are reusing the `SubsamplingScaleImageView`. If the user used pinch-to-zoom and panned around a previous page, the `SubsamplingScaleImageView` will retain those settings. So, we call `resetScaleAndCenter()` to switch back to the starting point, then call `setImage()` to hand the `SubsamplingScaleImageView` the page to render. `setImage()` takes an `ImageSource`, and in this case we need to use `cachedBitmap()`, to indicate both that the image is in the form of an existing `Bitmap` and that we are caching the `Bitmap` ourselves (so the `SubsamplingScaleImageView` should not attempt to `recycle()` the `Bitmap`).

The result is a PDF viewer, where we can pick a PDF from the "Open" action bar item, then swipe through the pages:

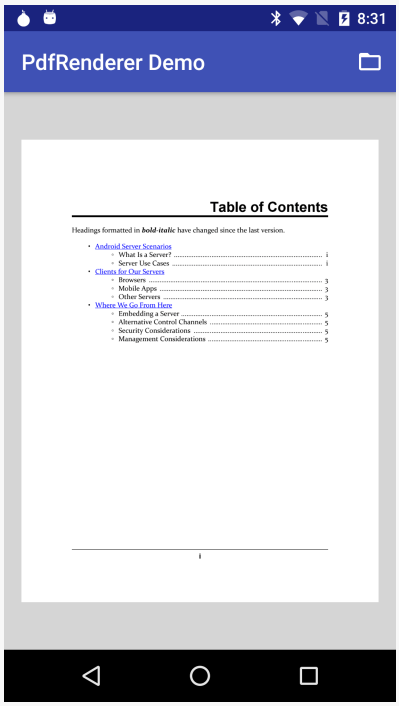


Figure 728: PdfRenderer Sample App

The Thunder Lizard Choice: PDF.js

`PdfRenderer` has many limitations, none bigger than the fact that it fails to render complex PDFs.

If you have used the Firefox Web browser on your desktop or notebook in the past few years, you may have noticed that its built-in PDF viewer is actually written in JavaScript, in the form of PDF.js. This is designed to handle more complex PDF files, and so it is a more complete solution than is `PdfRenderer`.

However, PDF.js requires a fairly robust Web rendering engine to work. On Android, that means we are limited to Android 4.4+, when the original `WebView` was replaced by the "Android System WebView" app's implementation, which shares more of its guts with Chromium.

The PDF/PdfJS sample project demonstrates the use of PDF.js.

PDF.js needs a `WebView`, so we put one in our layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView android:id="@+id/webview"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

(from PDF/PdfJS/app/src/main/res/layout/activity_main.xml)

Not surprisingly, it needs JavaScript to be enabled for that `WebView`. However, we also need to give JavaScript the ability to read from arbitrary URLs when we load the JavaScript itself from a `file` URL, as our PDF might come from somewhere else (e.g., `content` scheme for a document opened via `ACTION_OPEN_DOCUMENT`):

```
    wv=(WebView)findViewById(R.id.webview);
    wv.getSettings().setJavaScriptEnabled(true);
    wv.getSettings().setAllowUniversalAccessFromFileURLs(true);
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

PDF.js itself is stored in `assets/pdfjs/` in our `main` source set. This consists of the JavaScript library (`build/` directory) and the stock Web-based viewer wrapped around that library (`web/` directory). Combined, these two directories represent 4.6MB of material. While some of that could be stripped out or tweaked for mobile

use, this highlights one of the problems with the PDF.js solution: it is large. Those assets will compress somewhat — expect about 2MB added to your APK file.

This sample app now gives the "Open" action bar item a submenu, where the user can choose from two pre-packaged PDFs as assets (a set of presentation slides and the infamous 1040(A) tax form from the United States Internal Revenue Service) or to pick one from the filesystem:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/pdfs"
    android:icon="@drawable/ic_folder_open_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_pdf_files">
    <menu>
      <item android:id="@+id/preso" android:title="@string/menu_presentation"/>
      <item android:id="@+id/taxes" android:title="@string/menu_taxes" />
      <item android:id="@+id/open" android:title="@string/menu_open" />
    </menu>
  </item>
</menu>
```

(from PDF/PdfJS/app/src/main/res/menu/pdf.xml)

In `onOptionsItemSelected()` of `MainActivity`, we route to `loadPdf()` methods for the two assets and the `open()` method for the "Pick" option:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.preso) {
      loadPdf("MultiWindowAndYourApp.pdf");
      return(true);
    }
    else if (item.getItemId()==R.id.taxes) {
      loadPdf("f1040a.pdf");
      return(true);
    }
    else if (item.getItemId()==R.id.open) {
      open();
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

`open()` still uses `ACTION_OPEN_DOCUMENT` to allow the user to pick a PDF file. `onActivityResult()` still saves the `Uri` in the `pickedDocument` field but then calls a `loadPdfUri()` method with the string representation of that `Uri`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (resultCode==Activity.RESULT_OK) {
    pickedDocument=data.getData();
    loadPdfUri(pickedDocument.toString());
  }
}
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

Similarly, the `loadPdf()` method used by `onOptionsItemSelected()` for the assets stores the chosen asset name in a `chosenAsset` field, then calls `loadPdfUri()` with the proper `file:///android_asset/` URL:

```
private void loadPdf(String name) {
  chosenAsset=name;
  loadPdfUri("file:///android_asset/"+name);
}
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

`loadPdfUri()` then uses `loadUrl()` to load up the Web-based PDF viewer in assets, supplying the URL to the PDF in the `file` query parameter:

```
private void loadPdfUri(String uri) {
  try {
    wv.loadUrl("file:///android_asset/pdfjs/web/viewer.html?file="+
      URLEncoder.encode(uri, "UTF-8"));
  }
  catch (UnsupportedEncodingException e) {
    e.printStackTrace();
  }
}
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

This works "out of the box" for the assets, as both the Web viewer and the PDFs come from `file` URLs. To get the `content` scheme to work, you have to add `file://` to `HOSTED_VIEWER_ORIGINS` in `web/viewer.js`, to tell the Web viewer that `file://` is a valid origin for the viewer and that any reachable URL should be tried:

```
 var HOSTED_VIEWER_ORIGINS = [
  'null',
  'http://mozilla.github.io',
  'https://mozilla.github.io',
  'file://'
 ];
```

(from PDF/PdfJS/app/src/main/assets/pdfjs/web/viewer.js)

We also need to hold onto `chosenAsset` in our saved instance state:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putString(STATE_ASSET, chosenAsset);
  outState.putParcelable(STATE_PICKED, pickedDocument);
}
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

And we restore the PDF in `onCreate()`:

```
@SuppressLint("SetJavaScriptEnabled")
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  wv=(WebView)findViewById(R.id.webview);
  wv.getSettings().setJavaScriptEnabled(true);
  wv.getSettings().setAllowUniversalAccessFromFileURLs(true);

  if (savedInstanceState!=null) {
    chosenAsset=savedInstanceState.getString(STATE_ASSET);

    if (chosenAsset==null) {
      pickedDocument=savedInstanceState.getParcelable(STATE_PICKED);

      if (pickedDocument!=null) {
        loadPdfUri(pickedDocument.toString());
      }
    }
    else {
      loadPdf(chosenAsset);
    }
  }
}
```

(from PDF/PdfJS/app/src/main/java/com/commonsware/android/pdfjs/MainActivity.java)

The result is PDF.js's stock PDF viewer, in our `WebView`, where the user can scroll vertically to browse all the pages in the PDF:

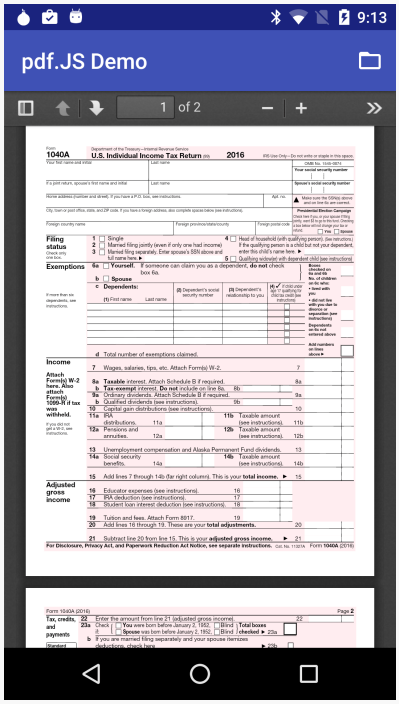


Figure 729: PDF.js Rendering IRS 1040 Form

The Native Approach: Pdfium

The major downside to both `PdfRenderer` and PDF.js as in-process PDF viewing solutions is the required API level. Both work with Android 5.0+, and PDF.js works with Android 4.4. However, you may have a `minSdkVersion` below 19. One approach would be to use an external PDF viewer for those older devices, but if that were an option, you may be better off using that for *all* Android versions, not just older ones.

Your remaining options involve using some C/C++ code for rendering PDFs.

One popular native code base for PDF rendering is Pdfium, from Google, used in Chromium and Chrome. Roughly speaking, it fills the same role there as PDF.js does with Firefox.

Bartosz Schiller's AndroidPdfViewer library wraps Pdfium in a View that handles rendering and standard gestures (e.g., horizontal swipes to move between pages).

On the plus side, Pdfium works well on older Android versions. The author of this book tested it back to Android 4.1 (API Level 16) and had no problems, and the library itself claims to support back to API Level 11.

However, there is a cost: APK size. By default, AndroidPdfViewer gives you NDK binaries that support the major CPU architectures: 32- and 64-bit ARM and x86, plus MIPS. As a result, the native binaries take up 30MB of space in your APK. Dropping support for CPU architectures that are less important to you (e.g., ARM) can help, and you can drop the per-APK cost to ~5MB if you use ABI splits and ship separate APKs per supported CPU architecture (on distribution channels where that is an option).

The PDF/Pdfium sample project demonstrates the use of AndroidPdfViewer and Pdfium. It is very similar to the PDF.js sample.

However, we need to pull in the AndroidPdfViewer library:

```
apply plugin: 'com.android.application'

android {
  compileSdkVersion 25
  buildToolsVersion '26.0.2'

  defaultConfig {
    applicationId "com.commonsware.android.pdfium"
    minSdkVersion 16
    targetSdkVersion 25
    versionCode 1
    versionName "1.0"
  }
}

dependencies {
  implementation 'com.github.barteksc:android-pdf-viewer:2.3.0'
}
```

(from PDF/Pdfium/app/build.gradle)

Our layout now uses a PDFView widget, instead of a RecyclerView or WebView:

```
<?xml version="1.0" encoding="utf-8"?>
<com.github.barteksc.pdfviewer.PDFView android:id="@+id/viewer"
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

(from PDF/Pdfium/app/src/main/res/layout/activity_main.xml)

We also need `open()` to support `ACTION_GET_CONTENT`, since `ACTION_OPEN_DOCUMENT` is not supported prior to API Level 19:

```
private void open() {
  if (Build.VERSION.SDK_INT<Build.VERSION_CODES.KITKAT) {
    Intent i=
      new Intent()
        .setType("application/pdf")
        .setAction(Intent.ACTION_GET_CONTENT)
        .addCategory(Intent.CATEGORY_OPENABLE);

    startActivityForResult(i, REQUEST_GET);
  }
  else {
    Intent i=
      new Intent()
        .setType("application/pdf")
        .setAction(Intent.ACTION_OPEN_DOCUMENT)
        .addCategory(Intent.CATEGORY_OPENABLE);

    startActivityForResult(i, REQUEST_OPEN);
  }
}
```

(from PDF/Pdfium/app/src/main/java/com/commonsware/android/pdfium/MainActivity.java)

Then, instead of `loadPdf()` or `loadPdfUri()` methods, we use a `configureViewer()` method. That method takes a `PDFView.Configurator` object, which we get by calling `fromUri()` on the `PDFView` itself, such as from our `onActivityResult()` method:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (resultCode==Activity.RESULT_OK) {
    pickedDocument=data.getData();
    chosenAsset=null;
    configureViewer(viewer.fromUri(pickedDocument));
  }
}
```

(from PDF/Pdfium/app/src/main/java/com/commonsware/android/pdfium/MainActivity.java)

`configureViewer()` then teaches the `PDFView` how we want to render this document:

```
private void configureViewer(PDFView.Configurator configurator) {
  configurator
    .enableSwipe(true)
    .swipeHorizontal(true)
    .enableDoubletap(true)
    .scrollHandle(new DefaultScrollHandle(this))
    .load();
}
```

(from PDF/Pdfium/app/src/main/java/com/commonsware/android/pdfium/MainActivity.java)

Here, we:

- Enable horizontal swiping to move between pages
- Enable a scroll handle as an alternative for moving between pages
- Enable a double-tap gesture for toggling between a few different zoom levels

The result is similar to the PDF.js result, since Pdfium can handle similarly-complex PDF files:

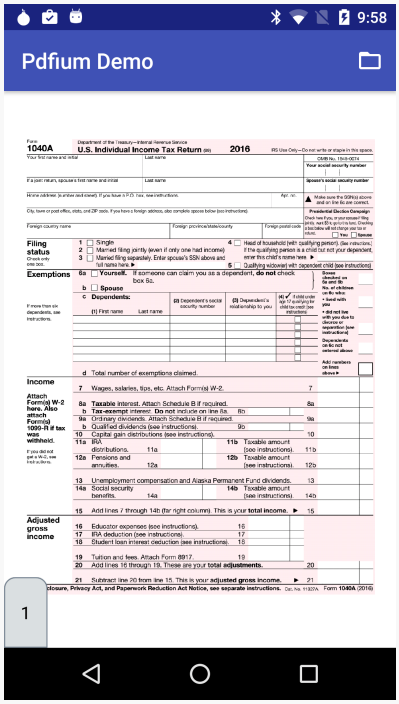


Figure 730: Pdfium Sample App Showing IRS 1040 Form

What To Choose?

So, what do you choose?

Option	Location Flexibility	Offline-Capable	PDF Complexity	Stability	Privacy	APK Size	`minSdkVersion`
External Viewer	good	good	varies	varies	poor	great	1
Google Docs	poor	poor	good	poor	poor	great	1
`PdfRenderer`	good	good	poor	good	great	great	21
PDF.js	good	good	good	good	great	meh	19
Pdfium	good	good	good	good	great	poor	11

For most apps, for most situations, using an external PDF viewer is the simplest solution. Use one that already exists, and be in position to suggest one to the user if there is no PDF viewer app available.

If you are certain that you need to keep the PDFs within your own app, and your `minSdkVersion` is 19 or higher, PDF.js offers the best rendering with a modest increase in app size. If, however, you need to support older devices than that, Pdfium will be a solid choice... but be prepared for some distribution complexity as you deal with the ABI splits to keep your APK size down to a reasonable level.

Trail: Home Screen Effects

Home Screen App Widgets

App widgets are live elements that the user can add to her home screen. Android ships with a variety of app widgets, such as a music player, and device manufacturers frequently add more. However, developers can add their own — in this chapter, we will see how this is done.

For the purposes of this book, "app widgets" will refer to these items that go on the home screen. Other uses of the term "widget" will be reserved for the UI widgets, subclasses of `View`, usually found in the `android.widget` Java package.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on:

- basic widgets
- broadcast `Intents`
- services

App Widgets and Security

Creating app widgets looks little like creating an activity. That is because the home screen is showing your app widget, whereas your own app shows your own activities. Having a third-party app (a home screen) show a UI from your app has some security ramifications.

Android's security model is based heavily on Linux user, file, and process security. Each application is (normally) associated with a unique user ID. All of its files are

owned by that user, and its process(es) run as that user. This prevents one application from modifying the files of another or otherwise injecting their own code into another running process. It would be dangerous for the home screen to run arbitrary code itself or somehow allow its UI to be directly manipulated by another process.

The app widget architecture, therefore, is set up to keep the home screen application independent from any code that puts app widgets on that home screen, so bugs in one cannot harm the other.

The Big Picture for a Small App Widget

The way Android pulls off this bit of security is through the use of `RemoteViews`.

The application component that supplies the UI for an app widget is not an `Activity`, but rather a `BroadcastReceiver` (often in tandem with a `Service`). The `BroadcastReceiver`, in turn, does not inflate a normal `View` hierarchy, like an `Activity` would, but instead inflates a layout into a `RemoteViews` object.

`RemoteViews` encapsulates a limited edition of normal widgets, in such a fashion that the `RemoteViews` can be "easily" transported across process boundaries. You configure the `RemoteViews` via your `BroadcastReceiver` and make those `RemoteViews` available to Android. Android in turn delivers the `RemoteViews` to the app widget host (usually the home screen), which renders them to the screen itself.

This architectural choice has many impacts:

- You do not have access to the full range of widgets and containers. You can use `FrameLayout`, `LinearLayout`, and `RelativeLayout` for containers, and `AnalogClock`, `Button`, `Chronometer`, `ImageButton`, `ImageView`, `ProgressBar`, and `TextView` for widgets. And, on API Level 11 and higher, you can use some `AdapterView`-based widgets, like `ListView`, as we will examine in the next chapter. And, as of API Level 16 (Android 4.1), you can use `GridLayout`... but not its backport on earlier devices.
- The only user input you can get is clicks of the `Button` and `ImageButton` widgets. In particular, there is no `EditText` for text input.
- Because the app widgets are rendered in another process, you cannot simply register an `OnClickListener` to get button clicks; rather, you tell `RemoteViews` a `PendingIntent` to invoke when a given button is clicked.

- You do not hold onto the `RemoteViews` and reuse them yourself. Rather, you create and send out a brand-new `RemoteViews` whenever you want to change the contents of the app widget. This, coupled with having to transport the `RemoteViews` across process boundaries, means that updating the app widget is expensive in terms of CPU time, memory, and battery life, when compared to equivalent UI updates of one of your own activities.
- Because the component handling the updates is a `BroadcastReceiver`, you have to be quick (lest you take too long and Android consider you to have timed out), you cannot use background threads, and your component itself is lost once the request has been completed. Hence, if your update might take a while, you will probably want to have the `BroadcastReceiver` start a `Service` and have the `Service` do the long-running task and eventual app widget update.

Crafting App Widgets

This will become somewhat easier to understand in the context of some sample code. In the `AppWidget/PairOfDice` project, you will find an app widget that displays a roll of a pair of dice. Clicking on the app widget re-rolls, in case you want a better result.

The Manifest

First, we need to register our `BroadcastReceiver` implementation in our `AndroidManifest.xml` file, along with a few extra features:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.appwidget.dice"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <uses-feature
    android:name="android.software.app_widgets"
    android:required="true"/>

  <application
```

```
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver
      android:name=".AppWidget"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>

    <activity
      android:name="PairOfDiceActivity"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from AppWidget/PairOfDice/app/src/main/AndroidManifest.xml)

Here, along with a do-nothing activity, we have a `<receiver>`. Of note:

1. Our `<receiver>` has `android:label` and `android:icon` attributes, which are not normally needed on `BroadcastReceiver` declarations. However, in this case, those are used for the entry that goes in the roster of available widgets to add to the home screen. Hence, you will probably want to supply values for both of those, and use appropriate resources in case you want translations for other languages.
2. Our `<receiver>` has an `<intent-filter>` for the `android.appwidget.action.APPWIDGET_UPDATE` action. This means we will get control whenever Android wants us to update the content of our app widget. There may be other actions we want to monitor — more on this in a later section.

3. Our `<receiver>` also has a `<meta-data>` element, indicating that its `android.appwidget.provider` details can be found in the `res/xml/widget_provider.xml` file. This metadata is described in greater detail shortly.

The uses-feature Element

If the central point of your application is to provide an app widget, you should strongly consider adding a `<uses-feature>` element to advertise this fact to markets like the Play Store:

```
<uses-feature android:name="android.software.app_widgets" android:required="true" />
```

In principle, having this element means that markets should block the installation of your app on devices where there is no app-widget-capable home screen or other known places for supporting app widgets.

If, however, your app has an app widget, but it is an adjunct to other forms of UI (typically a launcher activity), then you may wish to leave off this `<uses-feature>` element, or set it to `android:required="false"`.

The Metadata

Next, we need to define the app widget provider metadata. This has to reside at the location indicated in the manifest — in this case, in `res/xml/widget_provider.xml`:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="144dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:initialLayout="@layout/widget"
/>
```

(from AppWidget/PairOfDice/app/src/main/res/xml/widget_provider.xml)

Here, we provide a few pieces of information:

1. The minimum width and height of the app widget (`android:minWidth` and `android:minHeight`). These are approximate — the app widget host (e.g., home screen) will tend to convert these values into “cells” based upon the overall layout of the UI where the app widgets will reside. However, they should be no smaller than the minimums cited here. Also, ideally, you use

dip instead of px for the dimensions, so the number of cells will remain constant regardless of screen density.

2. The frequency in which Android should request an update of the widget's contents (android:updatePeriodMillis). This is expressed in terms of milliseconds, so a value of 3600000 is a 60-minute update cycle. Note that the minimum value for this attribute is 30 minutes — values less than that will be "rounded up" to 30 minutes. Hence our 15-minute (900000 millisecond) request will actually result in an update every 30 minutes.
3. The initial layout to use for the app widget, for the time between when the user requests the app widget and when onUpdate() of our AppWidgetProvider gets control.

Note that the calculations for determining the number of cells for an app widget varies. The dip dimension value for an N-cell dimension was (74 * N) - 2 (e.g., a 2x3 cell app widget would request a width of 146dip and a height of 220dip). The value as of API Level 14 (a.k.a., Ice Cream Sandwich) is now (70 * N) - 30 (e.g., a 2x3 cell app widget would request a width of 110dip and a height of 180dip). To have your app widgets maintain a consistent number of cells, you will need two versions of your app widget metadata XML, one in res/xml-v14/ (with the API Level 14 calculation) and one in res/xml/ (for prior versions of Android).

The Layout

Eventually, you are going to need a layout that describes what the app widget looks like. You need to stick to the widget and container classes noted above; otherwise, this layout works like any other layout in your project.

For example, here is the layout for the PairOfDice app widget:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/background"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/widget_frame"
    >
  <ImageView android:id="@+id/left_die"
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/die_5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
    android:layout_marginLeft="7dip"
  />
  <ImageView android:id="@+id/right_die"
    android:layout_centerVertical="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="7dip"
  />
</RelativeLayout>
```

(from AppWidget/PairOfDice/app/src/main/res/layout/widget.xml)

All we have is a pair of `ImageView` widgets (one for each die), inside of a `RelativeLayout`. The `RelativeLayout` has a background, specified as a nine-patch PNG file. This allows the `RelativeLayout` to have guaranteed contrast with whatever wallpaper is behind it, so the user can tell the actual app widget bounds.

The BroadcastReceiver

Next, we need a `BroadcastReceiver` that can get control when Android wants us to update our `RemoteViews` for our app widget. To simplify this, Android supplies an `AppWidgetProvider` class we can extend, instead of the normal `BroadcastReceiver`. This simply looks at the received `Intent` and calls out to an appropriate lifecycle method based on the requested action.

The one method that frequently needs to be implemented on the provider is `onUpdate()`. Other lifecycle methods may be of interest and are discussed later in this chapter.

For example, here is the implementation of the `AppWidgetProvider` for `PairOfDice`:

```
package com.commonsware.android.appwidget.dice;

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.widget.RemoteViews;

public class AppWidget extends AppWidgetProvider {
  private static final int[] IMAGES={R.drawable.die_1,R.drawable.die_2,
                                     R.drawable.die_3,R.drawable.die_4,
                                     R.drawable.die_5,R.drawable.die_6};
```

```
  @Override
  public void onUpdate(Context ctxt, AppWidgetManager mgr,
                       int[] appWidgetIds) {
    ComponentName me=new ComponentName(ctxt, AppWidget.class);

    mgr.updateAppWidget(me, buildUpdate(ctxt, appWidgetIds));
  }

  private RemoteViews buildUpdate(Context ctxt, int[] appWidgetIds) {
    RemoteViews updateViews=new RemoteViews(ctxt.getPackageName(),
                                            R.layout.widget);

    Intent i=new Intent(ctxt, AppWidget.class);

    i.setAction(AppWidgetManager.ACTION_APPWIDGET_UPDATE);
    i.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS, appWidgetIds);

    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0 , i,
                                                PendingIntent.FLAG_UPDATE_CURRENT);

    updateViews.setImageViewResource(R.id.left_die,
                                     IMAGES[(int)(Math.random()*6)]);
    updateViews.setOnClickPendingIntent(R.id.left_die, pi);
    updateViews.setImageViewResource(R.id.right_die,
                                     IMAGES[(int)(Math.random()*6)]);
    updateViews.setOnClickPendingIntent(R.id.right_die, pi);
    updateViews.setOnClickPendingIntent(R.id.background, pi);

    return(updateViews);
  }
}
```

(from AppWidget/PairOfDice/app/src/main/java/com/commonsware/android/appwidget/dice/AppWidget.java)

To update the `RemoteViews` for our app widget, we need to build those `RemoteViews` (delegated to a `buildUpdate()` helper method) and tell an `AppWidgetManager` to update the widget via `updateAppWidget()`. In this case, we use a version of `updateAppWidget()` that takes a `ComponentName` as the identifier of the widget to be updated. Note that this means that we will update all instances of this app widget presently in use — the concept of multiple app widget instances is covered in greater detail later in this chapter.

Working with `RemoteViews` is a bit like trying to tie your shoes while wearing mittens — it may be possible, but it is a bit clumsy. In this case, rather than using methods like `findViewById()` and then calling methods on individual widgets, we need to call methods on `RemoteViews` itself, providing the identifier of the widget we wish to modify. This is so our requests for changes can be serialized for transport to the home screen process. It does, however, mean that our view-updating code looks a fair bit different than it would if this were the main `View` of an activity or row of a `ListView`.

To create the `RemoteViews`, we use a constructor that takes our package name and the identifier of our layout. This gives us a `RemoteViews` that contains all of the widgets we declared in that layout, just as if we inflated the layout using a `LayoutInflater`. The difference, of course, is that we have a `RemoteViews` object, not a `View`, as the result.

We then use methods like:

1. `setImageViewResource()` to set the image for each of our `ImageView` widgets, in this case a randomly chosen die face (using graphics created from a set of SVG files from the OpenClipArt site)
2. `setOnClickPendingIntent()` to provide a `PendingIntent` that should get fired off when a die, or the overall app widget background, is clicked

We then supply that `RemoteViews` to the `AppWidgetManager`, which pushes the `RemoteViews` structure to the home screen, which renders our new app widget UI.

The Result

If you compile and install all of this, you will have a new app widget entry available. How you add app widgets to the home screen varies based upon Android version and the home screen implementation, and there are too many possibilities to try to list here.

No matter how you add the Pair of Dice, the app widget will appear on the home screen:

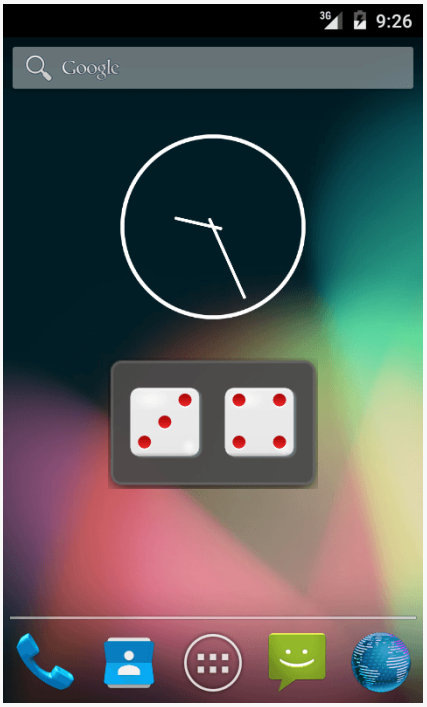


Figure 731: Pair of Dice, In Action

Another and Another

As indicated above, you can have multiple instances of the same app widget outstanding at any one time. For example, one might have multiple picture frames, or multiple "show-me-the-latest-RSS-entry" app widgets, one per feed. You will distinguish between these in your code via the identifier supplied in the relevant `AppWidgetProvider` callbacks (e.g., `onUpdate()`).

If you want to support separate app widget instances, you will need to store your state on a per-app-widget-identifier basis. You will also need to use an appropriate version of `updateAppWidget()` on `AppWidgetManager` when you update the app widgets, one that takes app widget identifiers as the first parameter, so you update the proper app widget instances.

Conversely, there is nothing requiring you to support multiple instances as independent entities. For example, if you add more than one `PairOfDice` app widget to your home screen, nothing blows up – they just show the same roll. That is because `PairOfDice` uses a version of `updateAppWidget()` that does not take any app widget IDs, and therefore updates all app widgets simultaneously.

App Widgets: Their Life and Times

There are three other lifecycle methods that `AppWidgetProvider` offers that you may be interested in:

1. `onEnabled()` will be called when the first widget instance is created for this particular widget provider, so if there is anything you need to do once for all supported widgets, you can implement that logic here
2. `onDeleted()` will be called when a widget instance is removed from the home screen, in case there is any data you need to clean up specific to that instance
3. `onDisabled()` will be called when the last widget instance for this provider is removed from the home screen, so you can clean up anything related to all such widgets

You will need to add appropriate action strings to your `<intent-filter>` for each of these events, such as `ACTION_APPWIDGET_ENABLED` to be notified about enabled events via `onEnabled()`.

Controlling Your (App Widget's) Destiny

As `PairOfDice` illustrates, you are not limited to updating your app widget only based on the timetable specified in your metadata. That timetable is useful if you can get by with a fixed schedule. However, there are cases in which that will not work very well:

1. If you want the user to be able to configure the polling period (the metadata is baked into your APK and therefore cannot be modified at runtime)
2. If you want the app widget to be updated based on external factors, such as a change in location

The recipe shown in `PairOfDice` will let you use `AlarmManager` (described in another chapter) or proximity alerts or whatever to trigger updates. All you need to do is:

1. Arrange for something to broadcast an `Intent` that will be picked up by the `BroadcastReceiver` you are using for your app widget provider
2. Have the provider process that `Intent` directly or pass it along to a `Service` (such as an `IntentService`)

Also, note that the `updatePeriodMillis` setting not only tells the app widget to update every so often, it will even *wake up the phone* if it is asleep so the widget can perform its update. On the plus side, this means you can easily keep your widgets up to date regardless of the state of the device. On the minus side, this will tend to drain the battery, particularly if the period is too fast. If you want to avoid this wakeup behavior, set `updatePeriodMillis` to `0` and use `AlarmManager` to control the timing and behavior of your widget updates.

Note that if there are multiple instances of your app widget on the user's home screen, they will all update approximately simultaneously if you are using `updatePeriodMillis`. If you elect to set up your own update schedule, you can control which app widgets get updated when, if you choose.

One Size May Not Fit All

It may be that you want to offer multiple app widget sizes to your users. Some might only want a small app widget. Some might really like what you have to offer and want to give you more home screen space to work in.

Android 1.x/2.x

The good news: this is easy to do.

The bad news: it requires you, in effect, to have one app widget per size.

The size of an app widget is determined by the app widget metadata XML file. That XML file is tied to a `<receiver>` element in the manifest representing one app widget. Hence, to have multiple sizes, you need multiple metadata files and multiple `<receiver>` elements.

This also means your app widgets will show up multiple times in the app widget selection list, when the user goes to add an app widget to their home screen. Hence, supporting many sizes will become annoying to the user, if they perceive you are "spamming" the app widget list. Try to keep the number of app widget sizes to a reasonable number (say, one or two sizes).

Android 3.0+

As of API Level 11, it is possible to have a resizeable app widget. To do this, you can have an `android:resizeMode` attribute in your widget metadata, with a value of

horizontal, vertical, or both (e.g., horizontal|vertical). When the user long-taps on an existing widget, they should see handles to allow the widget to be resized:

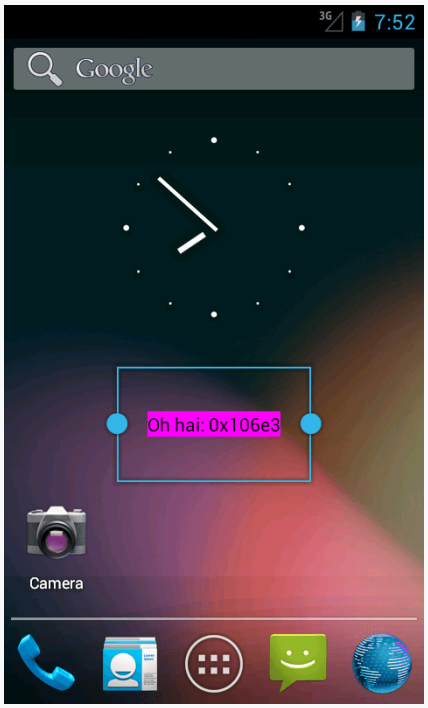


Figure 732: API Demos App Widget, Resizing

You can also have android:minResizeWidth and android:minResizeHeight attributes, measured in dp, that indicate the approximate smallest size that your app widget can support. These values will be interpreted in terms of "cells", as with the android:minWidth and android:minHeight attributes, and so the dp values you supply will not be used precisely.

However, for Android 3.x and 4.0 (API Level 11-15), your code would not be informed about being resized. You had to simply ensure that your layout would intelligently use any extra space automatically. Hence, resizing tended to be used primarily with adapter-driven app widgets, as will be discussed in the next chapter.

Starting with API Level 16, though, you can find out when the user resizes your app widget, so you can perhaps use a different layout for a different size, or otherwise adapt to the available space. Finding out about resize events takes a bit more work, as is illustrated in the AppWidget/Resize sample project.

This app widget project is similar to `PairOfDice`, described earlier in this chapter. However, our layout skips the dice, replacing them with a `TextView` widget in the `RelativeLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/background"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@drawable/widget_frame"
  android:orientation="horizontal">

  <TextView
    android:id="@+id/size"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:textColor="@android:color/white"
    android:textAppearance="?android:attr/textAppearanceMedium">

  </TextView>

</RelativeLayout>
```

(from AppWidget/Resize/app/src/main/res/layout/widget.xml)

Our `widget_provider.xml` resource stipulates our desired `android:resizeMode` and minimum resize dimensions:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="180dip"
  android:minHeight="110dip"
  android:minResizeWidth="110dip"
  android:minResizeHeight="40dip"
  android:initialLayout="@layout/widget"
  android:resizeMode="horizontal|vertical"
/>
```

(from AppWidget/Resize/app/src/main/res/xml/widget_provider.xml)

Finding out about app widget resizing is a different event than finding out about app widget updates. Hence, we need to add a new `<action>` element to the `<intent-filter>` of our `<receiver>` in the manifest, indicating that we want `APPWIDGET_OPTIONS_CHANGED` as well as `ACTION_UPDATE`:

```
    <receiver
      android:name="AppWidget"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
        <action android:name="android.appwidget.action.APPWIDGET_OPTIONS_CHANGED"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>
```

(from AppWidget/Resize/app/src/main/AndroidManifest.xml)

Then, our app widget implementation can override an `onAppWidgetOptionsChanged()` method:

```
@Override
public void onAppWidgetOptionsChanged(Context ctxt,
                                      AppWidgetManager mgr,
                                      int appWidgetId,
                                      Bundle newOptions) {
  RemoteViews updateViews=
      new RemoteViews(ctxt.getPackageName(), R.layout.widget);
  String msg=
      String.format(Locale.getDefault(),
                    "[%d-%d] x [%d-%d]",
                    newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MIN_WIDTH),
                    newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MAX_WIDTH),
                    newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MIN_HEIGHT),
                    newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MAX_HEIGHT));

  updateViews.setTextViewText(R.id.size, msg);

  mgr.updateAppWidget(appWidgetId, updateViews);
}
```

(from AppWidget/Resize/app/src/main/java/com/commonsware/android/appwidget/resize/AppWidget.java)

You will notice that we skip `onUpdate()`. We will be called with `onAppWidgetOptionsChanged()` when the app widget is added and resized. Hence, in the case of this app widget, we can define what the app widget looks like from `onAppWidgetOptionsChanged()`, avoiding `onUpdate()`. That being said, more typical app widgets will wind up implementing both methods, especially if they are supporting lower API levels than 16, where `onAppWidgetOptionsChanged()` will not be called.

Also remember that your process may well be terminated in between calls to app widget lifecycle methods like `onUpdate()` and `onAppWidgetOptionsChanged()`. Hence, if there is data from one method that you want in the other, be sure to persist that data somewhere.

In the `AppWidget` implementation of `onAppWidgetOptionsChanged()`, we can find out about our new app widget size by means of the `Bundle` supplied to our method. What we *cannot* find out is our exact size. Rather, we are provided minimum and maximum dimensions of our app widget via four values in the `Bundle`:

- `AppWidgetManager.OPTION_APPWIDGET_MIN_WIDTH`
- `AppWidgetManager.OPTION_APPWIDGET_MAX_WIDTH`
- `AppWidgetManager.OPTION_APPWIDGET_MIN_HEIGHT`
- `AppWidgetManager.OPTION_APPWIDGET_MAX_HEIGHT`

In our case, we grab these `int` values and pour them into a `String` template, using that to fill in the `TextView` of the app widget's contents.

When our app widget is initially launched, we show our initial size ranges:

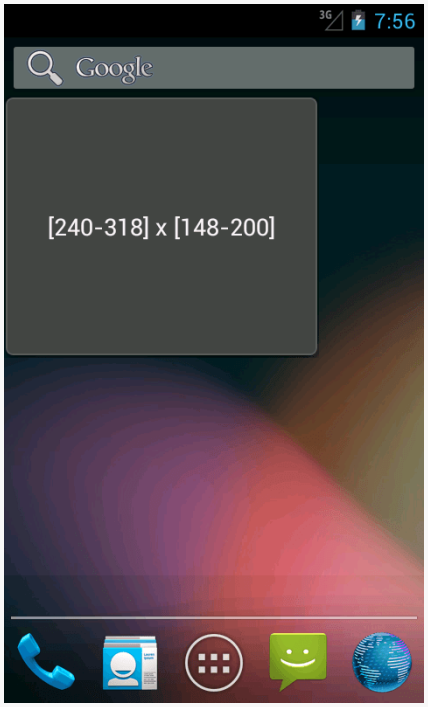


Figure 733: Resize Widget, As Initially Added

When the user resizes our app widget, we show the new size ranges:

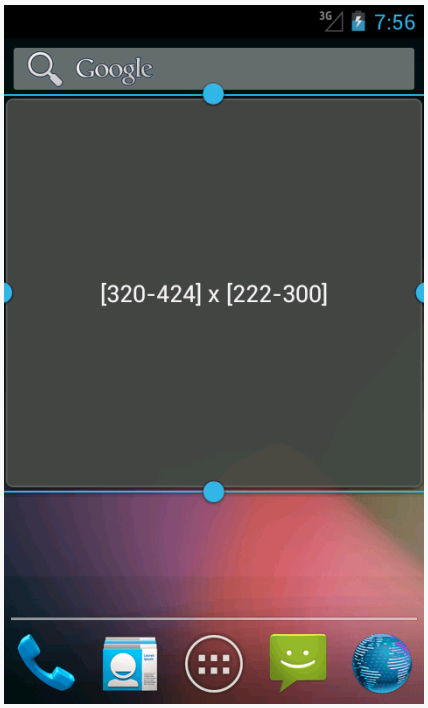


Figure 734: Resize Widget, During Resize Operation

However, not all home screen implementations will necessarily send the `APPWIDGET_OPTIONS_CHANGED` when an app widget is added to the home screen, only when the user resizes it later. For example, while the emulator's home screen for Android 4.1 broadcasts `APPWIDGET_OPTIONS_CHANGED`, it does not for 4.2 or 4.3. Hence, you may want to also examine the size information in `onUpdate()` as well, so that you react to the initial size as well as any future sizes. One way to do this is to simply iterate over the supplied app widget IDs and invoke your own `onAppWidgetOptionsChanged()` method:

```
// based on http://stackoverflow.com/a/18552461/115145

@Override
public void onUpdate(Context context,
                     AppWidgetManager appWidgetManager,
                     int[] appWidgetIds) {
  super.onUpdate(context, appWidgetManager, appWidgetIds);

  for (int appWidgetId : appWidgetIds) {
    Bundle options=appWidgetManager.getAppWidgetOptions(appWidgetId);
```

```
    onAppWidgetOptionsChanged(context, appWidgetManager, appWidgetId,
                              options);
  }
}
```

(from AppWidget/Resize/app/src/main/java/com/commonsware/android/appwidget/resize/AppWidget.java)

Lockscreen Widgets

Android's lockscreen (a.k.a., the keyguard) had long been unmodifiable by developers. This led to a number of developers creating so-called "replacement lockscreens", which generally reduce device security, as they can be readily bypassed. However, on Android 4.2 through 4.4, developers can create app widgets that the user can deploy to the lockscreen, helping to eliminate the need for "replacement lockscreens".

However, note that this capability was dropped with Android 5.0. As a result, this particular app widget feature may not be something that you want to worry about. That being said, it *is* available for those versions, and you are welcome to support it for those versions.

Declaring that an app widget supports being on the lockscreen instead of (or in addition to) the home screen is very easy. All you must do is add an `android:widgetCategory` attribute to your app widget metadata resource. That attribute should have a value of either `keyguard` (for the lockscreen), `home_screen`, or both (e.g., `keyguard|home_screen`), depending upon where you want the app widget to be eligible. By default, if this attribute is missing, Android assumes a default value of `home_screen`.

Users cannot resize the lockscreen widgets at this time. However, you still will want to specify an `android:resizeMode` attribute in your app widget metadata, as whether or not you include `vertical` resizing will affect the height of your app widget. Lockscreen widgets without `vertical` will have a fixed small height on tablets, while lockscreen widgets with `vertical` will fill the available height. Lockscreen widgets on phones will always be small (to fit above the PIN/password entry area), and lockscreen widgets on all devices will stretch to fill available space horizontally.

You can also specify a different starting layout to use when your app is added to the lockscreen, as opposed to being added to the home screen. To do this, just add an `android:initialKeyguardLayout` attribute to your app widget metadata, pointing to the lockscreen-specific layout to use.

To see this in action, take a look at the AppWidget/TwoOrThreeDice sample project. This is a revised clone of the PairOfDice sample, allowing the dice to be added to the lockscreen, and showing three dice on the lockscreen instead of the two on the home screen.

Our app widget metadata now contains the lockscreen-related attributes: android:widgetCategory and android:initialKeyguardLayout:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="144dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:initialLayout="@layout/widget"
  android:initialKeyguardLayout="@layout/lockscreen"
  android:widgetCategory="keyguard|home_screen"
/>
```

(from AppWidget/TwoOrThreeDice/app/src/main/res/xml/widget_provider.xml)

Our lockscreen layout simply adds a third die, middle_die:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/background"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/widget_frame"
    >
  <ImageView android:id="@+id/left_die"
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/die_3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
  />
  <ImageView android:id="@+id/middle_die"
    android:layout_centerInParent="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
    android:layout_marginRight="7dip"
  />
  <ImageView android:id="@+id/right_die"
```

```
    android:layout_centerVertical="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="7dip"
  />
</RelativeLayout>
```

(from AppWidget/TwoOrThreeDice/app/src/main/res/layout/lockscreen.xml)

However, by specifying a different layout for the lockscreen widget, we have a problem. We need to know, in our Java code, what layout to use for the `RemoteViews` and how many dice need to be updated. And, ideally, we would handle this in a backwards-compatible fashion, so our app widget will have its original functionality on older Android devices. Plus, supporting the lockscreen makes it that much more likely that the user will have more than one instance of our app widget (e.g., one on the lockscreen and one on the homescreen), so we should do a better job than `PairOfDice` did about handling multiple app widget instances.

To deal with the latter point, our new `onUpdate()` method iterates over each of the app widget IDs supplied to it and calls a private `updateWidget()` method for each, so we can better support multiple instances:

```
  @Override
  public void onUpdate(Context ctxt, AppWidgetManager mgr,
                       int[] appWidgetIds) {
    for (int appWidgetId : appWidgetIds) {
      updateWidget(ctxt, mgr, appWidgetId);
    }
  }
```

(from AppWidget/TwoOrThreeDice/app/src/main/java/com/commonsware/android/appwidget/dice/AppWidget.java)

The `updateWidget()` method is a bit more complicated than the `PairOfDice` equivalent code:

```
  @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
  private void updateWidget(Context ctxt, AppWidgetManager mgr,
                            int appWidgetId) {
    int layout=R.layout.widget;

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
      int category=
          mgr.getAppWidgetOptions(appWidgetId)
             .getInt(AppWidgetManager.OPTION_APPWIDGET_HOST_CATEGORY,
```

```
                    -1);

    layout=
        (category == AppWidgetProviderInfo.WIDGET_CATEGORY_KEYGUARD)
            ? R.layout.lockscreen : R.layout.widget;
  }

  RemoteViews updateViews=
      new RemoteViews(ctxt.getPackageName(), layout);
  Intent i=new Intent(ctxt, AppWidget.class);

  i.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);

  PendingIntent pi=
      PendingIntent.getBroadcast(ctxt, appWidgetId, i,
                                 PendingIntent.FLAG_UPDATE_CURRENT);

  updateViews.setImageViewResource(R.id.left_die,
                                   IMAGES[(int)(Math.random() * 6)]);
  updateViews.setOnClickPendingIntent(R.id.left_die, pi);
  updateViews.setImageViewResource(R.id.right_die,
                                   IMAGES[(int)(Math.random() * 6)]);
  updateViews.setOnClickPendingIntent(R.id.right_die, pi);
  updateViews.setOnClickPendingIntent(R.id.background, pi);

  if (layout == R.layout.lockscreen) {
    updateViews.setImageViewResource(R.id.middle_die,
                                     IMAGES[(int)(Math.random() * 6)]);
    updateViews.setOnClickPendingIntent(R.id.middle_die, pi);
  }

  mgr.updateAppWidget(appWidgetId, updateViews);
}
```

(from AppWidget/TwoOrThreeDice/app/src/main/java/com/commonsware/android/appwidget/dice/AppWidget.java)

First, we need to choose which layout we are working with. We assume that we are to use the original `R.layout.widget` resource by default. But, if we are on API Level 17 or higher, we can call `getAppWidgetOptions()` on the `AppWidgetManager`, to get the `Bundle` of options — the same options that we could be delivered in `onAppWidgetOptionsUpdate()` as described in the previous section. One value that will be in this `Bundle` is `AppWidgetManager.OPTION_APPWIDGET_HOST_CATEGORY`, which will be an `int` with a value of `AppWidgetProviderInfo.WIDGET_CATEGORY_KEYGUARD` if our app widget is on the lockscreen. In that case, we switch to using `R.layout.lockscreen`. In addition, we know then we need to update the `middle_die` when we are updating the other dice.

There is also a subtle change in our `getBroadcast()` call to `PendingIntent`: we pass in the app widget ID as the second parameter, whereas in `PairOfDice` we passed `0`. `PendingIntent` objects are cached in our process, and by default we will get the same `PendingIntent` when we call `getBroadcast()` for the same `Intent`. However, in our case, we may *want* two or more different `PendingIntent` objects for the same `Intent`, with differing extras (`EXTRA_APPWIDGET_ID`). Since extras are not considered when evaluating equivalence of `Intent` objects, just having different extras is insufficient to get different `PendingIntent` objects for those `Intent` objects. The second parameter to `getBroadcast()` (and `getActivity()` and `getService()`) on `PendingIntent` is a unique identifier, to differentiate between two otherwise equivalent `Intent` objects, forcing `PendingIntent` to give us distinct `PendingIntent` objects. This way, we can support two or more app widget instances, each having their own `PendingIntent` objects for their click events.

On an Android 4.2+ lockscreen, you should be able to swipe to one side (e.g., a bezel swipe from left to right), to expose an option to add an app widget:

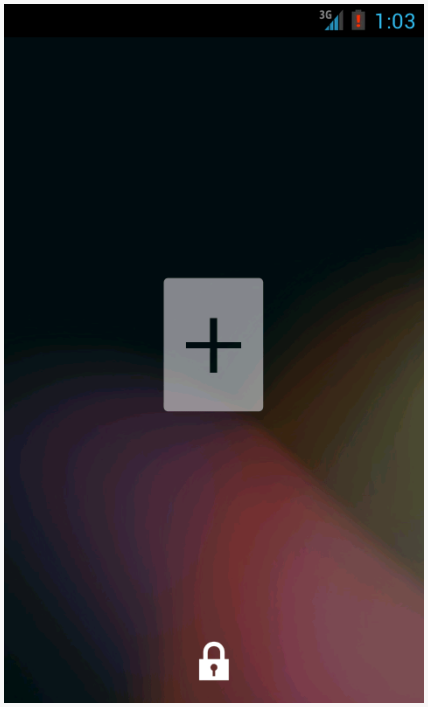


Figure 735: Lockscreen Add-A-Widget Panel, On a 4.2 Emulator

Tapping the "+" indicator (and, if needed, entering your device PIN or password), brings up an app widget chooser:

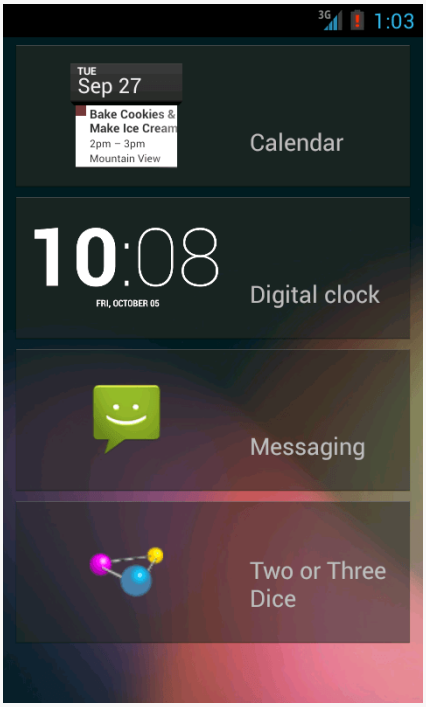


Figure 736: Lockscreen Widget Selection List, On a 4.2 Emulator

Choosing `TwoOrThreeDice` will then add the app widget to the lockscreen, with three dice, not two:

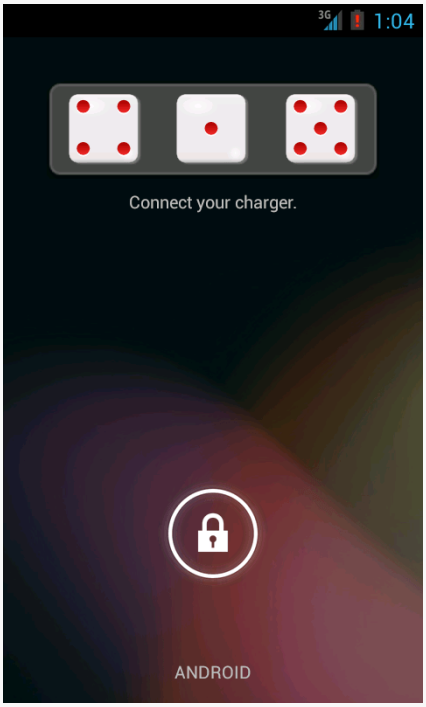


Figure 737: Lockscreen with TwoOrThreeDice, On a 4.2 Emulator

Preview Images

App widgets can have preview images attached. Preview images are drawable resources representing a preview of what the app widget might look like on the screen. On tablets, this will be used as part of an app widget gallery, replacing the simple context menu presentation you used to see on Android 1.x and 2.x phones:

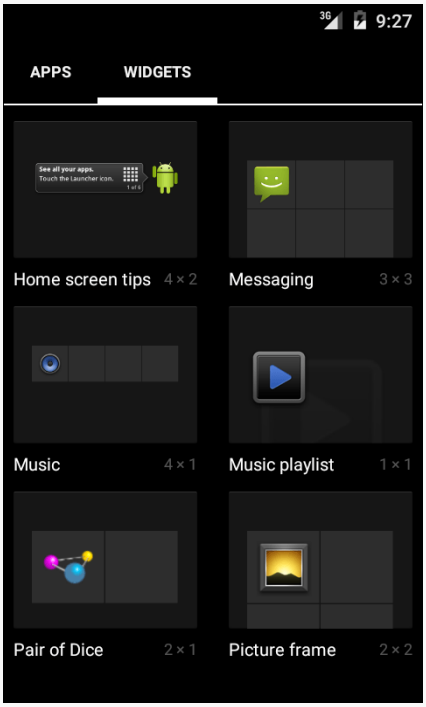


Figure 738: App Widget Gallery, on Android 5.0

To create the preview image itself, the Android 3.0+ emulator images contain a Widget Preview application that lets you run an app widget in its own container, outside of the home screen:

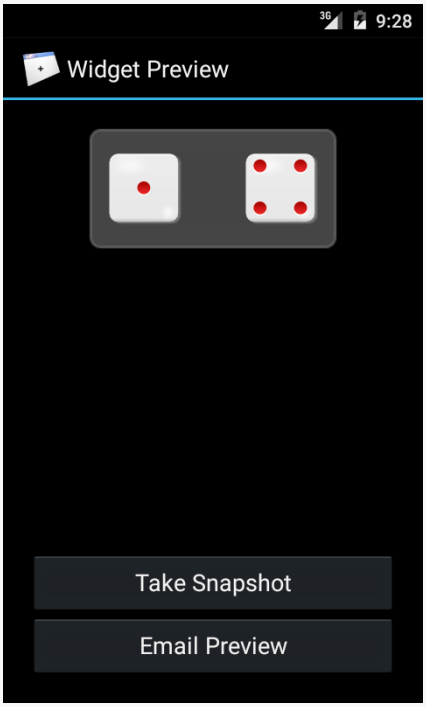


Figure 739: The Widget Preview application, showing a preview of the Analog Clock app widget

From here, you can take a snapshot and save it to external storage, copy it to your project's `res/drawable-nodpi/` directory (indicating that there is no intrinsic density assumed for this image), and reference it in your app widget metadata via an `android:previewImage` attribute. We will see an example of such an attribute in the chapter on advanced app widgets.

Being a Good Host

In addition to creating your own app widgets, it is possible to host app widgets. This is mostly aimed for those creating alternative home screen applications, so they can take advantage of the same app widget framework and all the app widgets being built for it.

This is not very well documented, but it involves the `AppWidgetHost` and `AppWidgetHostView` classes. The latter is a `View` and so should be able to reside in an app widget host's UI like any other ordinary widget.

Adapter-Based App Widgets

API Level 11 introduced a few new capabilities for app widgets, to make them more interactive and more powerful than before. The documentation lags a bit, though, so determining how to use these features takes a bit of exploring. Fortunately for you, the author did some of that exploring on your behalf, to save you some trouble.

Prerequisites

Understanding this chapter requires that you have read the preceding chapter and all of its prerequisites.

AdapterViews for App Widgets

In addition to the classic widgets available for use in app widgets and `RemoteViews`, five more were added for API Level 11:

1. `GridView`
2. `ListView`
3. `StackView`
4. `ViewFlipper`
5. `AdapterViewFlipper`

Three of these (`GridView`, `ListView`, `ViewFlipper`) are widgets that existed in Android since the outset. `StackView` was added in API Level 11 to provide a "stack of cards" UI:

Figure 740: The Google Books app widget, showing a StackView

`AdapterViewFlipper` works like a `ViewFlipper`, allowing you to toggle between various children with only one visible at a time. However, whereas with `ViewFlipper` all children are fully-instantiated `View` objects held by the `ViewFlipper` parent, `AdapterViewFlipper` uses the `Adapter` model, so only a small number of actual `View` objects are held in memory, no matter how many potential children there are.

With the exception of `ViewFlipper`, the other four all require the use of an `Adapter`. This might seem odd, as there is no way to provide an `Adapter` to a `RemoteViews`. That is true, but API Level 11 added new ways for `Adapter`-like communication between the app widget host (e.g., home screen) and your application. We will take an in-depth look at that in an upcoming section.

Building Adapter-Based App Widgets

In an activity, if you put a `ListView` or `GridView` into your layout, you will also need to hand it an `Adapter`, providing the actual row or cell `View` objects that make up the contents of those selection widgets.

In an app widget, this becomes a bit more complicated. The host of the app widget does not have any `Adapter` class of yours. Hence, just as we have to send the contents of the app widget's UI via a `RemoteViews`, we will need to provide the rows or cells via `RemoteViews` as well. Android, starting with API Level 11, has a `RemoteViewsService` and `RemoteViewsFactory` that you can use for this purpose.

Let's take a look, in the form of the AppWidget/LoremWidget sample project, which will put a ListView of 25 Latin words into an app widget.

The AppWidgetProvider

At its core, our AppWidgetProvider (named WidgetProvider, in a stunning display of creativity) still needs to create and configure a RemoteViews object with the app widget UI, then use updateAppWidget() to push that RemoteViews to the host via the AppWidgetManager. However, for an app widget that involves an AdapterView, like ListView, there are two more key steps:

- You have to tell the RemoteViews the identity of a RemoteViewsService that will help fill the role that the Adapter would in an activity
- You have to provide the RemoteViews with a "template" PendingIntent to be used when the user taps on a row or cell in the AdapterView, to replace the onListItemClick() or similar method you might have used in an activity

For example, here is WidgetProvider for our Latin-word app widget:

```
package com.commonsware.android.appwidget.lorem;

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.widget.RemoteViews;

public class WidgetProvider extends AppWidgetProvider {
  public static String EXTRA_WORD=
    "com.commonsware.android.appwidget.lorem.WORD";

  @Override
  public void onUpdate(Context ctxt, AppWidgetManager appWidgetManager,
                       int[] appWidgetIds) {
    for (int i=0; i<appWidgetIds.length; i++) {
      Intent svcIntent=new Intent(ctxt, WidgetService.class);

      svcIntent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);
      svcIntent.setData(Uri.parse(svcIntent.toUri(Intent.URI_INTENT_SCHEME)));

      RemoteViews widget=new RemoteViews(ctxt.getPackageName(),
                                         R.layout.widget);
```

```
      widget.setRemoteAdapter(R.id.words, svcIntent);

      Intent clickIntent=new Intent(ctxt, LoremActivity.class);
      PendingIntent clickPI=PendingIntent
                            .getActivity(ctxt, 0,
                                         clickIntent,
                                         PendingIntent.FLAG_UPDATE_CURRENT);

      widget.setPendingIntentTemplate(R.id.words, clickPI);

      appWidgetManager.updateAppWidget(appWidgetIds[i], widget);
    }

    super.onUpdate(ctxt, appWidgetManager, appWidgetIds);
  }
}
```

(from AppWidget/LoremWidget/app/src/main/java/com/commonsware/android/appwidget/lorem/WidgetProvider.java)

The call to `setRemoteAdapter()` is where we point the `RemoteViews` to our `RemoteViewsService` for our `AdapterView` widget. The main rules for the `Intent` used to identify the `RemoteViewsService` are:

1. The service must be identified by its data (`Uri`), so even if you create the `Intent` via the `Context`-and-`Class` constructor, you will need to convert that into a `Uri` via `toUri(Intent.URI_INTENT_SCHEME)` and set that as the `Uri` for the `Intent`. Why? While your application has access to your `RemoteViewsService` `Class` object, the app widget host will not, and so we need something that will work across process boundaries. You could elect to add your own `<intent-filter>` to the `RemoteViewsService` and use an `Intent` based on that, but that would make your service more publicly visible than you might want.
2. Any extras that you package on the `Intent` — such as the app widget ID in this case — will be on the `Intent` that is delivered to the `RemoteViewsService` when it is invoked by the app widget host.

Note that there are two flavors of `setRemoteAdapter()`. An older deprecated one takes the app widget ID as the first parameter. The current one does not. The current one, though, is only available on API Level 14 and higher.

The call to `setPendingIntentTemplate()` is where we provide a `PendingIntent` that will be used as the template for all row or cell clicks. As we will see in a bit, the

underlying `Intent` in the `PendingIntent` will have more data added to it by our `RemoteViewsFactory`.

In all other respects, our `WidgetProvider` is unremarkable compared to other app widgets. It will need to be registered in the manifest as a `<receiver>`, as with any other app widget.

The RemoteViewsService

Android supplies a `RemoteViewsService` class that you will need to extend, and this class is the one you must register with the `RemoteViews` for an `AdapterView` widget. For example, here is `WidgetService` (once again, a highly creative name) from the `LoremWidget` project:

```
package com.commonsware.android.appwidget.lorem;

import android.content.Intent;
import android.widget.RemoteViewsService;

public class WidgetService extends RemoteViewsService {
  @Override
  public RemoteViewsFactory onGetViewFactory(Intent intent) {
    return(new LoremViewsFactory(this.getApplicationContext(),
                                 intent));
  }
}
```

(from AppWidget/LoremWidget/app/src/main/java/com/commonsware/android/appwidget/lorem/WidgetService.java)

As you can see, this service is practically trivial. You have to override one method, `onGetViewFactory()`, which will return the `RemoteViewsFactory` to use for supplying rows or cells for the `AdapterView`. You are passed in an `Intent`, the one used in the `setRemoteAdapter()` call. Hence, if you have more than one `AdapterView` widget in your app widget, you could elect to have two `RemoteViewsService` implementations, or one that discriminates between the two widgets via something in the `Intent` (e.g., custom action string). In our case, we only have one `AdapterView`, so we create an instance of a `LoremViewFactory` and return it. Google has suggested using `getApplicationContext()` here to supply the `Context` object to `RemoteViewsFactory`, instead of using the `Service` as a `Context`, though it is unclear why this is.

Another thing different about the `RemoteViewsService` is how it is registered in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.appwidget.lorem"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-feature
    android:name="android.software.app_widgets"
    android:required="true"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="LoremActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <receiver
      android:name="WidgetProvider"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>

    <service
      android:name="WidgetService"
      android:permission="android.permission.BIND_REMOTEVIEWS"/>
  </application>

</manifest>
```

(from AppWidget/LoremWidget/app/src/main/AndroidManifest.xml)

Note the use of `android:permission`, specifying that whoever sends an `Intent` to `WidgetService` must hold the `BIND_REMOTEVIEWS` permission. This can only be held by the operating system. This is a security measure, so arbitrary applications cannot find out about your service and attempt to spoof being the OS and cause you to supply them with `RemoteViews` for the rows, as this might leak private data.

The RemoteViewsFactory

A `RemoteViewsFactory` interface implementation looks and feels a lot like an `Adapter`. In fact, one could imagine that the Android developer community might create `CursorRemoteViewsFactory` and `ArrayRemoteViewsFactory` and such to further simplify writing these classes.

For example, here is `LoremViewsFactory`, the one used by the `LoremWidget` project:

```
package com.commonsware.android.appwidget.lorem;

import android.appwidget.AppWidgetManager;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.widget.RemoteViews;
import android.widget.RemoteViewsService;

public class LoremViewsFactory implements
    RemoteViewsService.RemoteViewsFactory {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private Context ctxt=null;
  private int appWidgetId;

  public LoremViewsFactory(Context ctxt, Intent intent) {
    this.ctxt=ctxt;
    appWidgetId=
        intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                           AppWidgetManager.INVALID_APPWIDGET_ID);
  }

  @Override
  public void onCreate() {
    // no-op
  }
```

```
@Override
public void onDestroy() {
  // no-op
}

@Override
public int getCount() {
  return(items.length);
}

@Override
public RemoteViews getViewAt(int position) {
  RemoteViews row=
      new RemoteViews(ctxt.getPackageName(), R.layout.row);

  row.setTextViewText(android.R.id.text1, items[position]);

  Intent i=new Intent();
  Bundle extras=new Bundle();

  extras.putString(WidgetProvider.EXTRA_WORD, items[position]);
  extras.putInt(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
  i.putExtras(extras);
  row.setOnClickFillInIntent(android.R.id.text1, i);

  return(row);
}

@Override
public RemoteViews getLoadingView() {
  return(null);
}

@Override
public int getViewTypeCount() {
  return(1);
}

@Override
public long getItemId(int position) {
  return(position);
}

@Override
public boolean hasStableIds() {
  return(true);
}
```

```
  @Override
  public void onDataSetChanged() {
    // no-op
  }
}
```

(from AppWidget/LoremWidget/app/src/main/java/com/commonsware/android/appwidget/lorem/LoremViewsFactory.java)

You need to implement a handful of methods that have the same roles in a `RemoteViewsFactory` as they do in an `Adapter`, including:

1. `getCount()`
2. `getViewTypeCount()`
3. `getItemId()`
4. `hasStableIds()`

In addition, you have `onCreate()` and `onDestroy()` methods that you must implement, even if they do nothing, to satisfy the interface.

You will need to implement `getLoadingView()`, which will return a `RemoteViews` to use as a placeholder while the app widget host is getting the real contents for the app widget. If you return `null`, Android will use a default placeholder.

The bulk of your work will go in `getViewAt()`. This serves the same role as `getView()` does for an `Adapter`, in that it returns the row or cell `View` for a given position in your data set. However:

1. You have to return a `RemoteViews`, instead of a `View`, just as you have to use `RemoteViews` for the main content of the app widget in your `AppWidgetProvider`
2. There is no recycling, so you do not get a `View` (or `RemoteViews`) back to somehow repopulate, meaning you will create a new `RemoteViews` every time

The impact of the latter is that you do not want to put large data sets into an app widget, as scrolling may get sluggish, just as you do not want to implement an `Adapter` without recycling unused `View` objects.

In `LoremViewsFactory`, the `getViewAt()` implementation creates a `RemoteViews` for a custom row layout, cribbed from one in the Android SDK:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2006 The Android Open Source Project
```

```
    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

         http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
-->

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_vertical"
    android:paddingLeft="8dp"
    android:paddingStart="8dp"
    android:textColor="@android:color/white"
    android:minHeight="?android:attr/listPreferredItemHeight"
/>
```

(from AppWidget/LoremWidget/app/src/main/res/layout/row.xml)

Then, `getViewAt()` pours in a word from the static `String` array of Latin words into that `RemoteViews` for the `TextView` inside it. It also creates an `Intent` and puts the Latin word in as an `EXTRA_WORD` extra, then provides that `Intent` to `setOnClickFillInIntent()`. In addition, it adds the app widget instance ID as an extra, reusing the framework's own `AppWidgetManager.EXTRA_APPWIDGET_ID` as the key. The contents of the "fill-in" `Intent` are merged into the "template" `PendingIntent` from `setPendingIntentTemplate()`, and the resulting `PendingIntent` is what is invoked when the user taps on an item in the `AdapterView`. The fully-configured `RemoteViews` is then returned.

The Rest of the Story

The app widget metadata needs no changes related to `Adapter`-based app widget contents. However, `LoremWidget` does add the `android:previewImage` attribute:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="146dip"
```

```
  android:minHeight="146dip"
  android:updatePeriodMillis="0"
  android:initialLayout="@layout/widget"
  android:autoAdvanceViewId="@+id/words"
  android:previewImage="@drawable/preview"
  android:resizeMode="vertical"
/>
```

(from AppWidget/LoremWidget/app/src/main/res/xml/widget_provider.xml)

This points to the `res/drawable-nodpi/preview.png` file that represents a "widgetshot" of the app widget in isolation, obtained from the Widget Preview application:

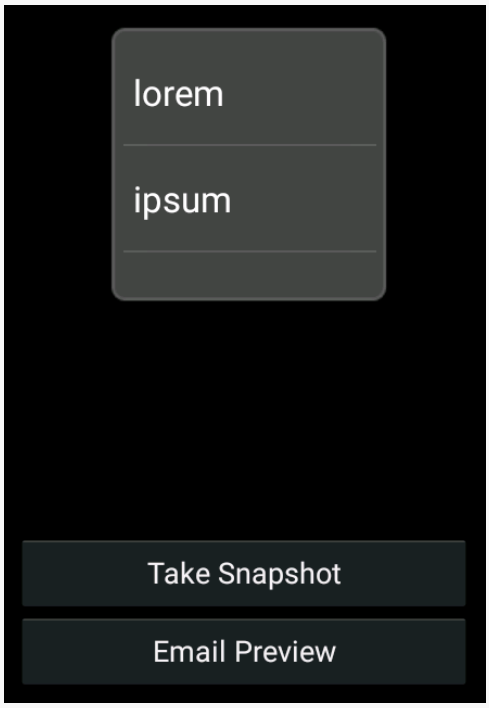


Figure 741: The preview of LoremWidget

Also, the metadata specifies `android:resizeMode="vertical"`. This attribute is new to Android 3.1, and allows the app widget to be resized by the user (in this case, only in the vertical direction, to show more rows). Older versions of Android will ignore this attribute, and the app widget will remain in your requested size. You can use `vertical`, `horizontal`, or both (via the pipe operator) as values for `android:resizeMode`.

When the user taps on an item in the list, our `PendingIntent` is set to bring up `LoremActivity`. This activity has `android:theme="@android:style/Theme.Translucent.NoTitleBar"` set in the manifest, meaning that it will not have its own user interface. Rather, it will extract our `EXTRA_WORD` — and the app widget ID — out of the `Intent` used to launch the activity and displays it in a `Toast` before finishing:

```
package com.commonsware.android.appwidget.lorem;

import android.app.Activity;
import android.appwidget.AppWidgetManager;
import android.os.Bundle;
import android.widget.Toast;

public class LoremActivity extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    String word=getIntent().getStringExtra(WidgetProvider.EXTRA_WORD);

    if (word == null) {
      word="We did not get a word!";
    }

    Toast.makeText(this,
                   String.format("#%d: %s",
                                 getIntent().getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                                                         AppWidgetManager.INVALID_APPWIDGET_ID),
                                 word), Toast.LENGTH_LONG).show();

    finish();
  }
}
```

(from AppWidget/LoremWidget/app/src/main/java/com/commonsware/android/appwidget/lorem/LoremActivity.java)

The Results

When you compile and install the application, nothing new shows up in the home screen launcher, because we have no activity defined to respond to `ACTION_MAIN` and `CATEGORY_HOME`. This would be unusual for an application distributed through the Play Store, as users often get confused if they install something and then do not know how to start it. However, for the purposes of this example, we should be fine, as readers of programming books never get confused about such things.

However, if you bring up the app widget gallery (e.g., long-tap on the home screen of an Android 6.0 device or emulator), you will see `LoremWidget` there, complete with preview image. You can drag it into one of the home screen panes and position it. When done, the app widget appears as expected:

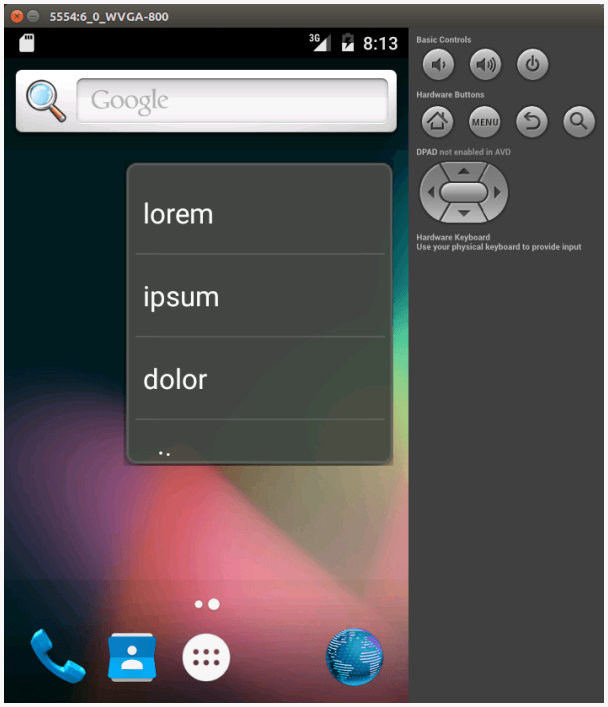


Figure 742: LoremWidget on Android Home Screen

The `ListView` is live and can be scrolled. Tapping an entry brings up the corresponding `Toast`.

If the user long-taps on the app widget, they will be able to reposition it. On Android 3.1 and beyond, when they lift their finger after the long-tap, the app widget will show resize handles on the sides designated by your `android:resizeMode` attribute:

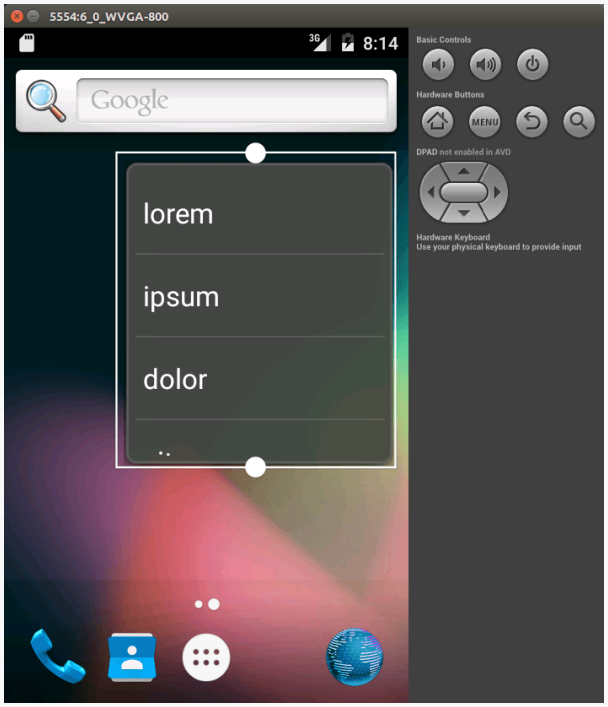


Figure 743: LoremWidget on Android Home Screen, with Resize Handles

The user can then drag those handles to expand or shrink the app widget in the specified dimensions:

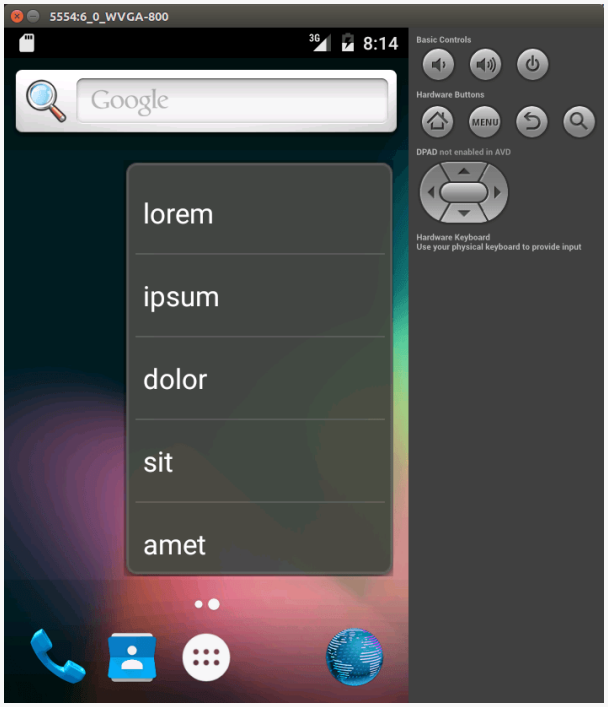


Figure 744: Resized LoremWidget on Android Home Screen

Advanced Permissions

Adding basic permissions to your app to allow it to, say, access the Internet, is fairly easy. However, the full permissions system has many capabilities beyond simply asking the user to let you do something. This chapter explores other uses of permissions, from securing your own components to using signature-level permissions (your own or Android's).

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on permissions and the chapter on signing your app.

One of the sample apps uses RxJava and RxAndroid, which are introduced elsewhere in the book.

Securing Yourself

Principally, at least initially, permissions are there to allow the user to secure their device. They have to agree to allow you to do certain things, such as reading contacts, that they might not appreciate.

The other side of the coin, of course, is to secure your own application. If your application is mostly activities, security may be just an "outbound" thing, where you request the right to use resources of other applications. If, on the other hand, you put content providers or services in your application, you will want to implement "inbound" security to control which applications can do what with the data.

Note that the issue here is less about whether other applications might "mess up" your data, but rather about privacy of the user's information or use of services that might incur expense. That is where the stock permissions for built-in Android applications are focused – can you read or modify contacts, can you send SMS, etc. If your application does not store information that might be considered private, security is less an issue. If, on the other hand, your application stores private data, such as medical information, security is much more important.

The first step to securing your own application using permissions is to declare said permissions, once again in the `AndroidManifest.xml` file. In this case, instead of `uses-permission`, you add `permission` elements. Once again, you can have zero or more `permission` elements, all as direct children of the root `manifest` element.

Declaring a permission is slightly more complicated than using a permission. There are three pieces of information you need to supply:

- The symbolic name of the permission. To keep your permissions from colliding with those from other applications, you should use your application's Java namespace as a prefix
- A label for the permission: something short that would be understandable by users
- A description for the permission: something a wee bit longer that is understandable by your users

```
<permission
  android:name="vnd.tlagency.sekrits.SEE_SEKRITS"
  android:label="@string/see_sekrits_label"
  android:description="@string/see_sekrits_description" />
```

This does not enforce the permission. Rather, it indicates that it is a possible permission; your application must still flag security violations as they occur.

Enforcing Permissions via the Manifest

There are two ways for your application to enforce permissions, dictating where and under what circumstances they are required. The easier one is to indicate in the manifest where permissions are required.

Activities, services, and receivers can all declare an attribute named `android:permission`, whose value is the name of the permission that is required to access those items:

```
<activity
  android:name=".SekritApp"
  android:label="Top Sekrit"
  android:permission="vnd.tlagency.sekrits.SEE_SEKRITS">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
      android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
```

Only applications that have requested your indicated permission will be able to access the secured component. In this case, "access" means:

1. Activities cannot be started without the permission
2. Services cannot be started, stopped, or bound to an activity without the permission
3. Intent receivers ignore messages sent via `sendBroadcast()` unless the sender has the permission

Enforcing Permissions Elsewhere

In your code, you have two additional ways to enforce permissions.

Your services can check permissions on a per-call basis via `checkCallingPermission()`. This returns `PERMISSION_GRANTED` or `PERMISSION_DENIED` depending on whether the caller has the permission you specified. For example, if your service implements separate read and write methods, you could require separate read versus write permissions in code by checking those methods for the permissions you need from Java.

Also, you can include a permission when you call `sendBroadcast()`. This means that eligible broadcast receivers must hold that permission; those without the permission are ineligible to receive it. We will examine `sendBroadcast()` in greater detail elsewhere in this book.

Requiring Standard System Permissions

While normally you require your own custom permissions using the techniques described above, there is nothing stopping you from reusing a standard system permission, if it would fit your needs.

For example, suppose that you are writing YATC (Yet Another Twitter Client). You decide that in addition to YATC having its own UI, you will design YATC to be a "Twitter engine" for use by third party apps:

- Send timeline updates via broadcast `Intents`
- Publish the timeline, the user's own tweets, @-mentions, and the like via a `ContentProvider`
- Offer a command-based service interface for posting updates to the timeline
- And so on

You could, and perhaps should, implement your own custom permission. However, since any app can get to Twitter just by having the `INTERNET` permission, one could argue that a third-party app should just need that same `INTERNET` permission to use your API (rather than integrating JTwitter or another third-party JAR).

Signature Permissions

Each permission in Android is assigned a protection level, via an `android:protectionLevel` attribute on the `<permission>` element. By default, permissions are at a `normal` level, but they can also be flagged as `dangerous`, `signatureOrSystem`, or `signature`. In the latter two cases, "signature" means that the app requesting the permission and the app requiring the permission should have be signed by the same signing key. In the case of `signatureOrSystem` — only used by the firmware – the app requesting the permission either needs to be signed by the firmware's signing key or reside on the system partition (e.g., come pre-installed with the device).

Firmware-Only Permissions

Most of Android's permissions mentioned in this book are ones that any SDK application can hold, if they ask for them and the user grants them. `INTERNET`, `READ_CONTACTS`, `ACCESS_FINE_LOCATION`, and kin all are normal permissions.

`BRICK` is not.

There was a permission in Android, named `BRICK`, that, in theory, allows an application to render a phone inoperable (a.k.a., "brick" the phone). While there is no `brickMe()` method in the Android SDK tied to this permission, presumably there might be something deep in the firmware that was protected by this permission.

Though, since Android 6.0 removed the `BRICK` permission from the SDK, it is clearly not something Google expects us to use.

The `BRICK` permission could not be held by ordinary Android SDK applications. You could request it all you want, and it will not be granted.

However, applications that are signed with the same signing key that signed the firmware *could* hold the `BRICK` permission.

That is because the system's own manifest used to have the following `<permission>` element:

```
<permission android:name="android.permission.BRICK"
        android:label="@string/permlab_brick"
        android:description="@string/permdesc_brick"
        android:protectionLevel="signature" />
```

Your Own Signature Permissions

You too can require `signature`-level permissions. That will restrict the holders of that permission to be other apps signed by your signing key. This is particularly useful for inter-process communication between apps in a suite — by using `signature` permissions, you ensure that only your apps will be able to participate in those communications.

One nice thing about these sorts of signature-level permissions is that the user is not bothered with them. It is assumed that the user will agree to the communication between the apps signed by the same signing key. Hence, the user will not see signature-level permissions at install or upgrade time.

Since in some cases, you may not be sure which app will be installed first, it is best to have all apps in the suite include the *same* `<permission>` element, in addition to the corresponding `<uses-permission>` element. That way, no matter which app is installed first, it can declare the permission that all will share.

Though, that has its own problems, as you will see in the next section.

The Custom Permission Vulnerability

(NOTE: Some of the material in this section originally appeared in material hosted in the CWAC-Security project repository. In addition, the author would like to thank Mark Carter and "Justin Case" for their contributions in this topic area).

Unfortunately, custom permissions have some undocumented limitations that make them intrinsically risky. Specifically, custom permissions can be defined by anyone, at any time, and "first one in wins", which opens up the possibility of unexpected behavior.

Here, we will walk through some scenarios and show where the problems arise, plus discuss how to mitigate them as best we can.

Scenarios

All of the following scenarios focus on three major app profiles.

App A is an app that defines a custom permission in its manifest, such as:

```
<permission
  android:name="com.commonsware.cwac.security.demo.OMG"
  android:description="@string/perm_desc"
  android:label="@string/perm_label"
  android:protectionLevel="normal"/>
```

App A also defends a component using the `android:permission` attribute, referencing the custom permission:

```
<provider
  android:name="FileProvider"
  android:authorities="com.commonsware.cwac.security.demo.files"
  android:exported="true"
  android:grantUriPermissions="false"
  android:permission="com.commonsware.cwac.security.demo.OMG">
  <grant-uri-permission android:path="/test.pdf"/>
</provider>
```

App B has a `<uses-permission>` element to declare to the user that it wishes to access components defended by that permission:

```
<uses-permission android:name="com.commonsware.cwac.security.demo.OMG"/>
```

App C has the same <uses-permission> element. The difference is that App B *also* has the <permission> element, just as App A does, albeit with different descriptive information (e.g., android:description) and, at times, a different protection level.

All three apps are signed with different signing keys, because in the real world they would be from different developers.

So, to recap:

- A defines a permission and uses it for defense
- B defines the same permission and requests to hold it
- C just requests to hold this permission

With all that in mind, let's walk through some possible scenarios, focusing on two questions:

1. What is the user told, when the app is installed through normal methods (i.e., not via adb), regarding this permission?
2. What access, if any, does App B or App C have to the ContentProvider from App A?

The Application SDK Case (A, Then C)

Suppose the reason why App A has defined a custom permission is because it wants third-party apps to have the ability to access its secured components... but only with user approval. By defining a custom permission, and having third-party apps request that permission, the user should be informed about the requested permission and can make an informed decision.

Conversely, if an app tries to access a secured component but has not requested the permission, the access attempt should fail.

App C has requested the custom permission via the <uses-permission> element. If the permission — defined by App A — has an android:protectionLevel of normal or dangerous, the user will be informed about the requested permission at install time. If the user continues with the installation, App C can access the secured component.

If, however, the android:protectionLevel is signature, the user is not informed about the requested permission at install time, as the system can determine on its own whether or not the permission should be granted. In this case, App A and App

C are signed with different signing keys, so Android silently ignores the permission request. If the user continues with installation, then App C tries to access App A's secured component, App C crashes with a `SecurityException`.

In other words, this all works as expected.

The Application SDK Problem Case (C, Then A)

However, in many cases, there is nothing forcing the user to install App A before App C. This is particularly true for publicly-distributed apps on common markets, like the Play Store.

When the user installs App C, the user is not informed about the request for the custom permission, presumably because that permission has not yet been defined. If the user later installs App A, App C is not retroactively granted the permission, and so App C's attempts to use the secured component fail.

This works as expected, though it puts a bit of a damper on custom permissions. One way to work around this would be for the user to uninstall App C, then install it again (with App A already installed). This returns us to the original scenario from the preceding section. However, if the user has data in App C, losing that data may be a problem (as in a "let's give App C, or perhaps App A, one-star ratings on the Play Store" sort of problem).

The Peer Apps Case, Part One (A, Then B)

Suppose now we augment our SDK-consuming app (formerly App C) to declare the same permission that App A does, in an attempt to allow the two apps to be installed in either order. That is what App B is: the same app as App C, but where it has the same `<permission>` element as does App A in its manifest.

This scenario is particularly important where both apps could be of roughly equal importance to the user. In cases where App C is some sort of plugin for App A, it is not unreasonable for the author of App A to require App A to be installed first. But, if Twitter and Facebook wanted to access components of each others' apps, it would be unreasonable for either of those firms to mandate that their app must be installed first. After all, if Twitter wants to be installed first, and Facebook wants to be installed first, one will be disappointed.

If the user installs App A (the app defending a component with the custom permission) before App B, the user will be notified at install time about App B's request for this permission. Notably, the information shown on the installation security screen will contain App A's description of the permission. And, if the user goes ahead and installs App B, App B can indeed access App A's secured component, since it was granted permission by the user.

Once again, everything is working as expected. Going back to the two questions:

1. The user is informed when App B or App C requests the permission defined by App A.
2. App B and App C can hold that permission if and only if they meet the requirements of the protection level

The Peer Apps Case, Part Two (B, Then A)

What happens if we reverse the order of installation? After all, if App A and App B are peers, from the standpoint of the user, there is roughly a 50% chance that the user will install App B before App A.

Here is where things go off the rails.

The user is not informed about App B's request for the custom permission.

The user will be informed about any platform permissions that the app requests via other `<uses-permission>` elements. If there are none, the user is told that App B requests no permissions... despite the fact that it does.

When the user installs App A, the same thing occurs. Of course, since App A does not have a `<uses-permission>` element, this is not all that surprising.

However, at this point, **even though the user was not informed**, App B holds the custom permission and can access the secured component.

This is bad enough when both parties are ethical. App B could be a piece of malware, though, designed to copy the data from App A, ideally without the user's knowledge. And, if App B is installed before App A, that would happen.

So, going to the two questions:

1. The user is **not** informed about App B's request for the permission...

2. ...but App B gets it anyway and can access the secured component

The Downgraded-Level Malware Case (B, Then A, Again)

You might think that the preceding problem would only be for `normal` or `dangerous` protection levels. If App A defines a permission as requiring a matching `signature`, and App A marks a component as being defended by that permission, Android must require the signature match, right?

Wrong.

The behavior is identical to the preceding case. Android does not use the *defender's* protection level. It uses the *definer's* protection level, meaning the protection level of whoever was installed first and had the `<permission>` element.

So, if App A has the custom permission defined as `signature`, and App B has the custom permission defined as `normal`, if App B is installed first, the behavior is as shown in the preceding section:

1. The user is **not** informed about App B's request for the permission...
2. ...but App B gets it anyway and can access the secured component, despite the signatures not matching

The Peer Apps Case With a Side Order of C

What happens if we add App C back into the mix? Specifically, what if App B is installed first, then App A, then App C?

When App C eventually gets installed, the user is prompted for the custom permission that App C requests via `<uses-permission>`. However, the description that the user sees is from App B, the one that first defined the custom `<permission>`. Moreover, the protection level is whatever App B defined it to be. So if App B downgraded the protection level from App A's intended `signature` to be `normal`, App C can hold that permission and access the secured App A component, even if it is signed by another signing key.

Not surprisingly, the same results occur if you install App B, then App C, then App A.

Behavior Analysis

The behavior exhibited in these scenarios is consistent with two presumed implementation "features" of Android's permission system:

1. First one in wins. In other words, the first app (or framework, in the case of the OS's platform permissions) that defines a `<permission>` for a given `android:name` gets to determine what the description is and what the protection level is.
2. The user is only prompted to confirm a permission if the app being installed has a `<uses-permission>` element, the permission was already defined by some other app, and the protection level is not `signature`.

Risk Assessment

The "first one in wins" rule is a blessing and a curse. It is a curse, insofar as it opens up the possibility for malware to hold a custom permission without the user's awareness of that, and even to downgrade a `signature`-level permission to `normal`. However, it is a blessing, in that the malware would have to be installed first; if it is installed second, either its request to hold the permission will be seen by the user (`normal` or `dangerous`) or the request to hold the permission will be rejected (`signature`).

This makes it somewhat unlikely for a piece of malware to try to sneakily make off with data. Eventually, if enough users start to ask publicly why App B needs access to App A's data (for cases where App A was installed first and the user knows about the permission request), somebody in authority may eventually realize that this is a malware attack. Of course, "eventually" may be a rather long time.

However, there are some situations where Android's custom permission behavior presents risk even greater than that. If the attacker has a means of being *sure* that their app was installed first, they can hold any permission from any third-party app they want to that was known at install time.

For example:

- Somebody could sell a used Android device, and the buyer could neglect to factory-reset it, and the malware could be installed by the seller
- Somebody could sell a used Android device with a ROM mod preinstalled, based on a normal ROM mod (e.g., CyanogenMod), but with an additional bit of malware installed, to prevent a factory reset from foiling the attack'

- Somebody could distribute devices to users who might think the device is "factory clean" and not laden with malware (e.g., devices given as gifts)
- Somebody could distribute devices to users who might think that the pre-installed malware is actually a legitimate app (e.g., devices given to employees by an employer wishing to monitor usage by examining protected data from third-party apps)

Android 5.0's "Fix"

Android 5.0 now prevents two apps from defining the same `<permission>` ("same" based on `android:name`) unless they are signed by the same signing key. First one in wins; the second app installation will fail.

On the plus side:

- This solves the security problem, as an attacker (B) cannot get at a defender's (A's) data by virtue of having been installed first, as A simply cannot be installed in this case.
- This has no impact on developers using `signature`-level `<permission>` elements for their own app suite.

However, it does pose significant limitations on legitimate public uses of custom `<permission>` elements. Only the defender should have the `<permission>` element now. Some client of the defender's app (C) should not have the `<permission>` element and should simply rely upon the fact that the defender should be installed first. If the client were to define the `<permission>`, then either the client *or* the defender cannot be installed, which is pointless.

This has usability issues:

- A client should check, on first run of their app, if an expected defender (and its `<permission>` element) exists. If not, the client should alert the user to this fact and perhaps stop the app from proceeding further. The user would have to uninstall the client, install the defender, then reinstall the client, to get everything working properly, and the more the user uses the client app, the more painful the uninstall might be.
- It is impossible for two apps to be clients of each other. By definition, one app has to be installed first and the other second, which means only the first-to-be-installed app can have a custom `<permission>`. If Facebook wanted to hold a custom Twitter permission, and Twitter wanted to hold a custom Facebook permission, one of them is out of luck — if Facebook is

installed first, it cannot request Twitter's permission (as it does not yet exist) nor can it define Twitter's permission (as if it does, Twitter cannot be installed). This *might* be able to be overcome for apps that are pre-loaded as part of a ROM mod or other custom Android build.

And, of course, this fix is only for Android 5.0 and above.

Mitigation Using PermissionUtils

The "first one in wins" rule also leads us to a mitigation strategy: On first run of our app, see if any other app has defined permissions that we have defined. If that has happened, then we are at risk, and take appropriate steps. If, however, no other app has defined our custom permissions, then the Android permission system should work for us, and we can proceed as normal.

The CWAC-Security library provides some helper code, in the form of the `PermissionUtils` class, to detect other apps defining the same custom permissions that you define.

The idea is that you call `checkCustomPermissions()` — a static method on `PermissionUtils` — on the first run of your app. It will return details about what other apps have already defined custom permissions that your app defines. If `checkCustomPermissions()` returns nothing, you know that everything is fine, and you can move ahead. Otherwise, you can:

- Check to see if the offending app is on some whitelist, or otherwise meets criteria that suggests that it is OK
- Alert the user, indicating that these already-installed apps will have access to your app secured components
- Upload details about the offending apps to your server, so you can try to track down whether they are legitimate users of some API that you are exposing or are malware
- Whatever else you feel is necessary

Custom Dangerous Permissions, and Android 6.0

Android 6.0 introduced the concept of runtime permissions, where `dangerous` permissions need to be requested at runtime in addition to being requested in the manifest. This is covered back in the introductory chapter on permissions.

However, what happens if you define a custom `dangerous` permission?

The good news is that it works. However, you will want to test it, in part to see what it looks like to users, so you can get the phrasing of your permission-related string resources correct.

The `Permissions/CustomDangerous` sample project contains two application modules:

- `app` is an app that defends an activity using a custom `dangerous` permission
- `client` is an app that wishes to request that permission and start that protected activity

The `<permission>` element is unremarkable, other than the `protectionLevel` being set to `dangerous`:

```
<permission
  android:name="com.commonsware.android.perm.custdanger.SOMETHING"
  android:description="@string/perm_desc"
  android:label="@string/perm_label"
  android:protectionLevel="dangerous" />
```

(from Permissions/CustomDangerous/app/src/main/AndroidManifest.xml)

The label and description come from string resources:

```
<string name="perm_label">Custom Dangerous Permission</string>
<string name="perm_desc">This is a description. No, really.</string>
```

(from Permissions/CustomDangerous/app/src/main/res/values/strings.xml)

The `client` module uses the same `AbstractPermissionActivity` seen elsewhere in this book to request that `com.commonsware.android.perm.custdanger.SOMETHING` permission at runtime.

If you install the `app` application, then install and run the `client` application, what you see is the *description*, not the label, appear in the runtime permission dialog:

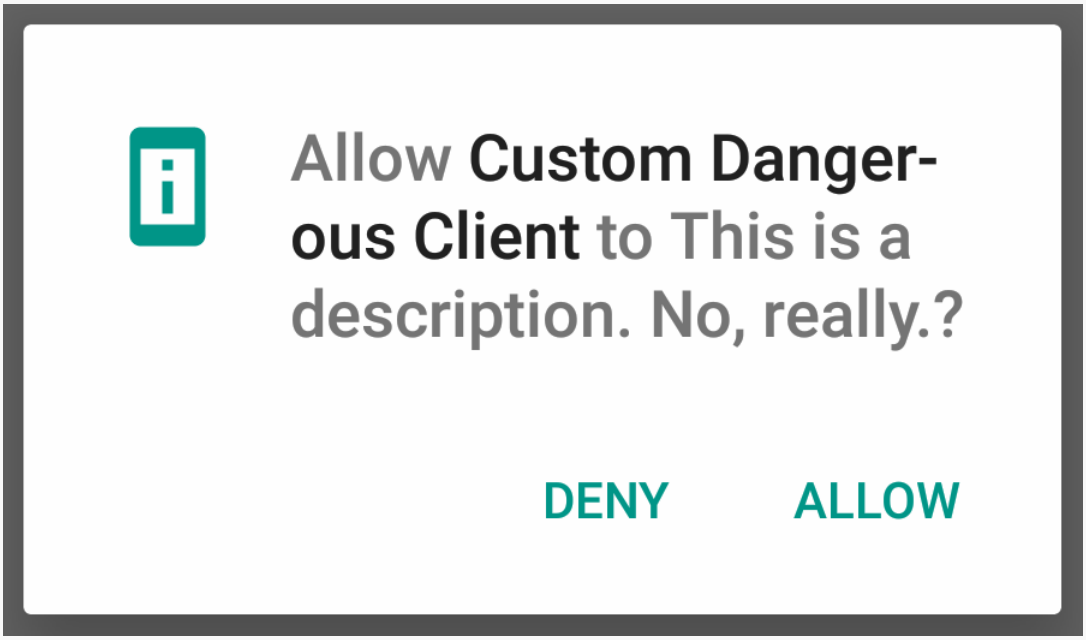


Figure 745: Custom Dangerous Permission, As Shown In Runtime Permission Dialog, on Android 6.0.1

If you go into Settings > Apps > Custom Dangerous Client > Permissions, the custom `dangerous` permission does not show up immediately, due to a poorly-designed UI:

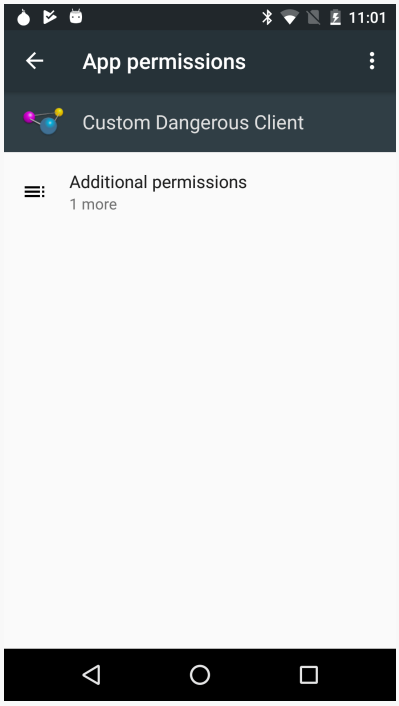


Figure 746: App Permissions in Settings, Showing Nothing Useful

Instead, the user needs to tap on the "Additional permissions" row to have that be replaced by the custom `dangerous` permission:

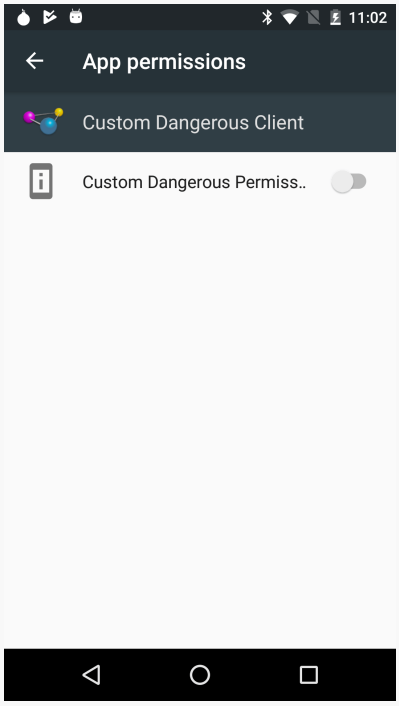


Figure 747: App Permissions in Settings, Showing Custom Dangerous Permission

Here, the *label* is what shows up, not the description.

Hence, you will want to tailor your phrasing of these string resources to make sense in their respective use cases.

Finding the Available Permissions

On the one hand, developers should try to stick to documented permissions.

On the other hand, documentation is sometimes lacking. This is particularly true for permissions other than those defined by the OS itself, ones that come from other apps that change more frequently, including the Play Services SDK and framework.

You might find that you need to determine what permissions have been defined on a given device. Perhaps that need is at runtime — if you request a permission that does not exist, you cannot actually get it, and that may lead to problems in the future. Perhaps that need is just during development itself, to inspect some device and determine what it does and does not have in terms of permissions.

PackageManager offers methods to allow you to examine the device's permissions and permission groups. The Permissions/PermissionReporter sample app uses these methods to build up a tabbed UI listing the defined permissions, broken down by protection level.

PackageManager and Permission Groups

getAllPermissionGroups() on PackageManager will return a list of PermissionGroupInfo objects. This method takes an int value; 0 generally will be fine for your use cases.

On its own, PermissionGroupInfo is not especially useful. However, you can turn around and call queryPermissionsByGroup() on PackageManager, passing in the name from the PermissionGroupInfo, to get all of the permissions in that group. This method also takes an int value as the second parameter, where once again 0 will be fine.

queryPermissionsByGroup() returns a List of PermissionInfo objects. PermissionInfo has a few interesting values:

- name, which is the fully-qualified name of the permission
- descriptionRes, which is the string resource ID from the permission's android:description attribute
- protectionLevel, which is a set of flags indicating the nature of the permission's security

Note that to get the actual text of the description, there is a loadDescription() method on PermissionInfo that will do all the work to find the actual string for the description, based upon the app that defined the permission and the current locale.

To get the details of all the permissions defined on a device, we will have to call queryPermissionsByGroup() for each permission group. Each of those calls will involve IPC, and so this *might* be slow enough to warrant its own thread.

With that in mind, MainActivity in the PermissionReporter sample app has a PermissionSource that collects information about the permissions on the system. That information is aggregated in a PermissionRoster, which the data emitted by this Observable:

```
    private Context ctxt;
```

```
  private PermissionSource(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
  }

  @Override
  public void subscribe(ObservableEmitter<PermissionRoster> emitter)
    throws Exception {
    PackageManager pm=ctxt.getPackageManager();
    final PermissionRoster result=new PermissionRoster();

    addPermissionsFromGroup(pm, null, result);

    for (PermissionGroupInfo group :
      pm.getAllPermissionGroups(0)) {
      addPermissionsFromGroup(pm, group.name, result);
    }

    emitter.onNext(result);
    emitter.onComplete();
  }

  private void addPermissionsFromGroup(PackageManager pm,
                                       String groupName,
                                       PermissionRoster result)
    throws PackageManager.NameNotFoundException {
    for (PermissionInfo info :
      pm.queryPermissionsByGroup(groupName, 0)) {
      int coreBits=
        info.protectionLevel &
          PermissionInfo.PROTECTION_MASK_BASE;

      switch (coreBits) {
        case PermissionInfo.PROTECTION_NORMAL:
          result.add(PermissionType.NORMAL, info);
          break;

        case PermissionInfo.PROTECTION_DANGEROUS:
          result.add(PermissionType.DANGEROUS, info);
          break;

        case PermissionInfo.PROTECTION_SIGNATURE:
          result.add(PermissionType.SIGNATURE, info);
          break;

        default:
          result.add(PermissionType.OTHER, info);
          break;
      }
```

```
      }
    }
  }
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/MainActivity.java)

The `subscribe()` method loops over the permission groups:

```
        pm.getAllPermissionGroups(0)) {
        addPermissionsFromGroup(pm, group.name, result);
      }
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/MainActivity.java)

As it turns out, not all permissions are part of a group. To find out the details of these un-grouped permissions, you need to call `queryPermissionsByGroup()` with a `null` permission group name.

For each permission group (plus the magic `null` group), we call a private `addPermissionsFromGroup()` method to collect the details of the permissions in that group:

```
                                      String groupName,
                                      PermissionRoster result)
      throws PackageManager.NameNotFoundException {
      for (PermissionInfo info :
        pm.queryPermissionsByGroup(groupName, 0)) {
        int coreBits=
          info.protectionLevel &
            PermissionInfo.PROTECTION_MASK_BASE;

        switch (coreBits) {
          case PermissionInfo.PROTECTION_NORMAL:
            result.add(PermissionType.NORMAL, info);
            break;

          case PermissionInfo.PROTECTION_DANGEROUS:
            result.add(PermissionType.DANGEROUS, info);
            break;

          case PermissionInfo.PROTECTION_SIGNATURE:
            result.add(PermissionType.SIGNATURE, info);
            break;

          default:
```

```
            result.add(PermissionType.OTHER, info);
            break;
        }
      }
    }
  }
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/MainActivity.java)

The `protectionLevel` field on a `PermissionInfo` contains a number of different sorts of flags. The `PROTECTION_MASK_BASE` is a bitmask that restricts the bits we are looking at to the ones for basic protections. We then divide the permissions into four groups based on protection level:

- normal
- dangerous
- signature
- other (which, on older devices, will include `system` or `signatureOrSystem` permissions)

Those `PermissionInfo` objects are then poured into the `PermissionRoster` object:

```
package com.commonsware.android.permreporter;

import android.content.pm.PermissionInfo;
import android.os.Parcel;
import android.os.Parcelable;
import java.util.ArrayList;
import java.util.HashMap;

class PermissionRoster {
  private HashMap<PermissionType, ArrayList<PermissionInfo>> roster=
      new HashMap<PermissionType, ArrayList<PermissionInfo>>();

  void add(PermissionType type, PermissionInfo info) {
    ArrayList<PermissionInfo> list=roster.get(type);

    if (list==null) {
      list=new ArrayList<>();
      roster.put(type, list);
    }

    list.add(info);
  }

  ArrayList<PermissionInfo> getListForType(PermissionType type) {
```

```
    return(roster.get(type));
  }
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionRoster.java)

`PermissionType` is an `enum` defined by this project for the four groups and includes some Java shenanigans for being able to convert back and forth between integer values and the `enum` values:

```
package com.commonsware.android.permreporter;

import android.util.SparseArray;

enum PermissionType {
  NORMAL(0),
  DANGEROUS(1),
  SIGNATURE(2),
  OTHER(3);

  private static final SparseArray<PermissionType> BY_VALUE=
      new SparseArray<PermissionType>(4);

  static {
    for (PermissionType type : PermissionType.values()) {
      BY_VALUE.put(type.value, type);
    }
  }

  private final int value;

  PermissionType(int value) {
    this.value=value;
  }

  static PermissionType forValue(int value) {
    return(BY_VALUE.get(value));
  }
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionType.java)

The Rest of the Sample

Of course, having a `PermissionRoster` and `PermissionSource` in isolation is not especially useful. Something needs to use RxJava to load these permissions. That is handled by the UI layer.

The Activity and ViewPager

The `MainActivity` has a `ViewPager`, along with a third-party tab implementation. `onCreate()` uses RxJava to get the `PermissionRoster` from the `PermissionSource`.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  final ViewPager pager=(ViewPager)findViewById(R.id.pager);
  final MaterialTabs tabs=(MaterialTabs)findViewById(R.id.tabs);

  observable=(Observable<PermissionRoster>)getLastNonConfigurationInstance();

  if (observable==null) {
    observable=Observable
      .create(new PermissionSource(this))
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .cache();
  }

  sub=observable.subscribe(new Consumer<PermissionRoster>() {
      @Override
      public void accept(PermissionRoster roster) throws Exception {
        pager.setAdapter(new PermissionTabAdapter(MainActivity.this,
          getFragmentManager(), roster));
        tabs.setViewPager(pager);
      }
    }, new Consumer<Throwable>() {
      @Override
      public void accept(Throwable error) throws Exception {
        Toast
          .makeText(MainActivity.this, error.getMessage(), Toast.LENGTH_LONG)
          .show();
        Log.e(getClass().getSimpleName(), "Exception processing request",
          error);
      }
```

```
    });
  }
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/MainActivity.java)

`PermissionTabAdapter` sets up four tabs, one per `PermissionType`, with labels pulled from string resources, and with instances of `PermissionListFragment` as the tab contents:

```
package com.commonsware.android.permreporter;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;

public class PermissionTabAdapter extends FragmentPagerAdapter {
  private static final int[] TITLES={
      R.string.normal,
      R.string.dangerous,
      R.string.signature,
      R.string.other};
  private final Context ctxt;
  private final PermissionRoster roster;

  PermissionTabAdapter(Context ctxt, FragmentManager mgr,
                       PermissionRoster roster) {
    super(mgr);

    this.ctxt=ctxt;
    this.roster=roster;
  }

  @Override
  public int getCount() {
    return(4);
  }

  @Override
  public Fragment getItem(int position) {
    PermissionType type=PermissionType.forValue(position);

    return(PermissionListFragment.newInstance(roster.getListForType(type)));
  }

  @Override
  public String getPageTitle(int position) {
```

```
    return(ctxt.getString(TITLES[position]));
  }
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionTabAdapter.java)

The `PermissionListFragment` instances are provided the `PermissionType` associated with their position in the `ViewPager`, courtesy of the `forValue()` lookup method implemented on `PermissionType`

The Tab Content

`PermissionListFragment` uses the factory method pattern to hold onto that `PermissionType` in the arguments `Bundle`, so it survives a configuration change:

```
private static final String ARG_PERMS="perms";

static PermissionListFragment newInstance(ArrayList<PermissionInfo> perms) {
  PermissionListFragment frag=new PermissionListFragment();
  Bundle args=new Bundle();

  args.putParcelableArrayList(ARG_PERMS, perms);
  frag.setArguments(args);

  return(frag);
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionListFragment.java)

The `PermissionListFragment` registers for events on the event bus in `onResume()` and unregisters in `onPause()`:

```
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  ArrayList<PermissionInfo> perms=
    getArguments().getParcelableArrayList(ARG_PERMS);

  if (perms!=null && perms.size()>0) {
    Collections.sort(perms, new Comparator<PermissionInfo>() {
      @Override
      public int compare(PermissionInfo one, PermissionInfo two) {
        return (one.name.compareTo(two.name));
      }
    });
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionListFragment.java)

By using sticky events with greenrobot's EventBus, we do not have to worry about configuration changes, as we can pick up the last-delivered `PermissionRosterLoadedEvent` after the change.

Speaking of that event, there is an `onLoadMainThread()` method to pick up that `PermissionRosterLoadedEvent`, sort the permissions by name, and populate the `ListView` associated with this `ListFragement`:

```
      setListAdapter(new PermissionAdapter(perms));
    }
    else {
      setListAdapter(new PermissionAdapter(new ArrayList<PermissionInfo>()));
      setEmptyText(getActivity().getString(R.string.msg_no_perms));
    }
  }

  private class PermissionAdapter extends ArrayAdapter<PermissionInfo> {
    PermissionAdapter(ArrayList<PermissionInfo> perms) {
      super(getActivity(), android.R.layout.simple_list_item_1, perms);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View result=super.getView(position, convertView, parent);
      TextView tv=(TextView)result.findViewById(android.R.id.text1);

      tv.setText(getItem(position).name);
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionListFragment.java)

The permissions are shown via a `PermissionAdapter`, which just uses the permission name for the contents:

```
      return(result);
    }
  }
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/PermissionListFragment.java)

This is required because `PermissionInfo` lacks a useful `toString()` implementation, so a simple `ArrayAdapter` is insufficient.

A Bit More Rx

`MainActivity` might undergo a configuration change, during or after we have loaded the permission information. So, we retain the `Observable` that we create around the `PermissionSource` via `onRetainNonConfigurationInstance()`:

```
public Object onRetainNonConfigurationInstance() {
  return(observable);
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/MainActivity.java)

In `onCreate()`, we only create the `Observable` if we received `null` from `getLastNonConfigurationInstance()`. In this fashion, we retain the `Observable` – and its cached output — across the configuration change.

We also `dispose()` of our subscription in `onDestroy()`, since that cannot (and should not) be retained:

```
protected void onDestroy() {
  sub.dispose();

  super.onDestroy();
}
```

(from Permissions/PermissionReporter/app/src/main/java/com/commonsware/android/permreporter/MainActivity.java)

The Results

Running the app gives you four tabs for the four different `PermissionType` values:

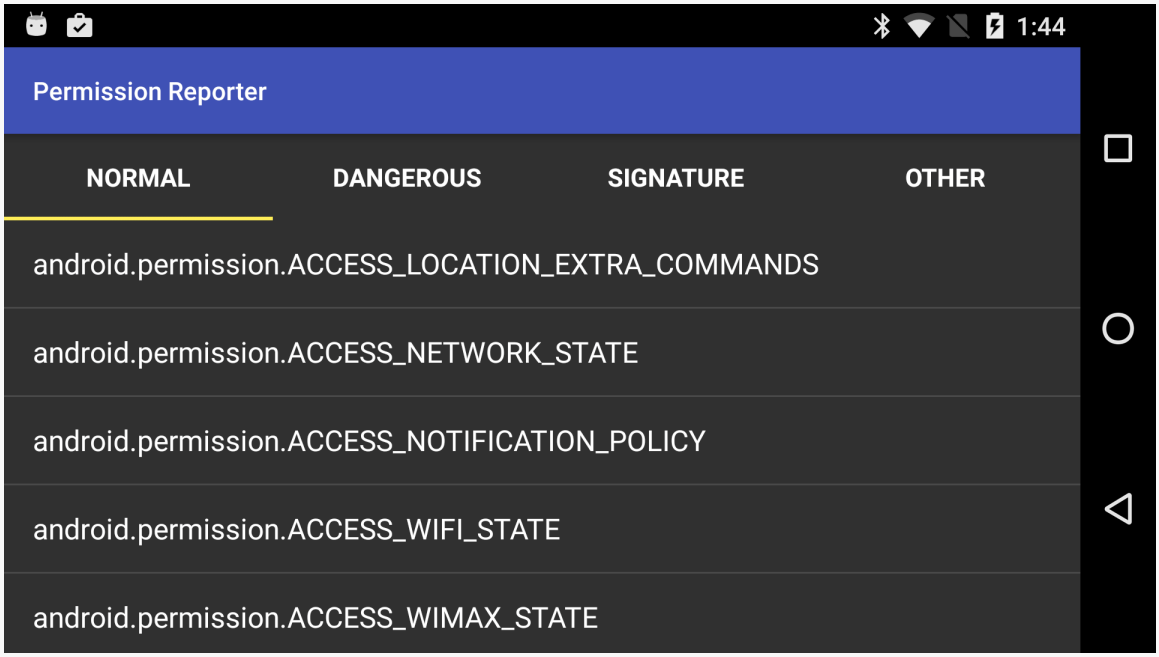


Figure 748: PermissionReporter, Normal Tab, on Android 6.0

The fourth tab, for "other" permissions, is typically empty:

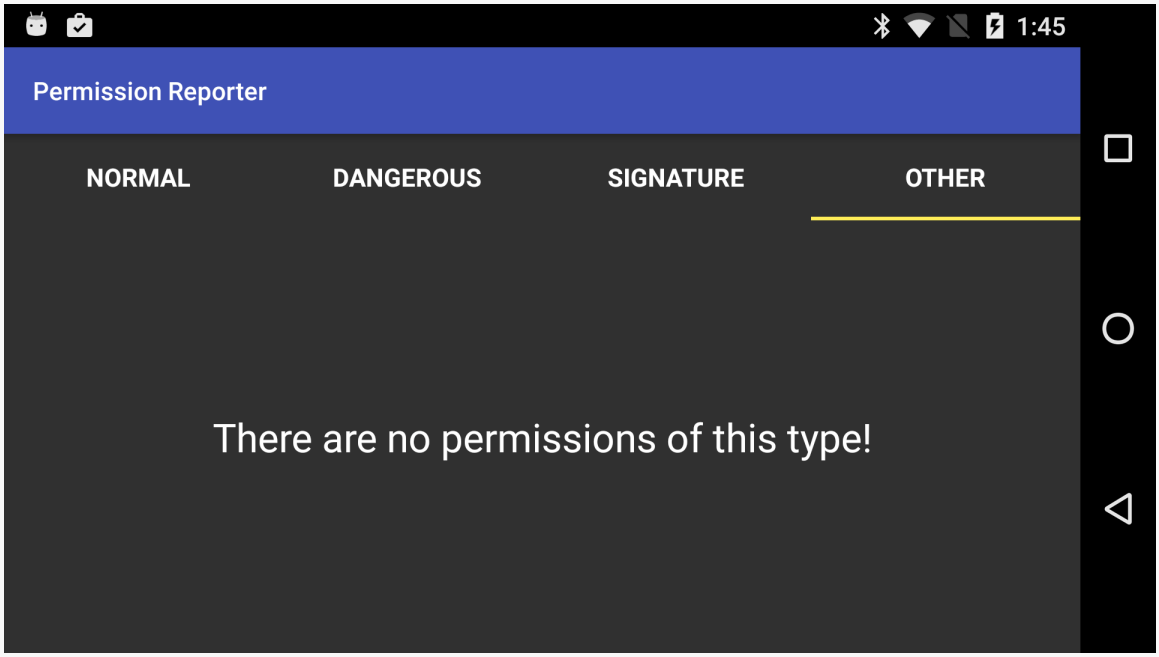
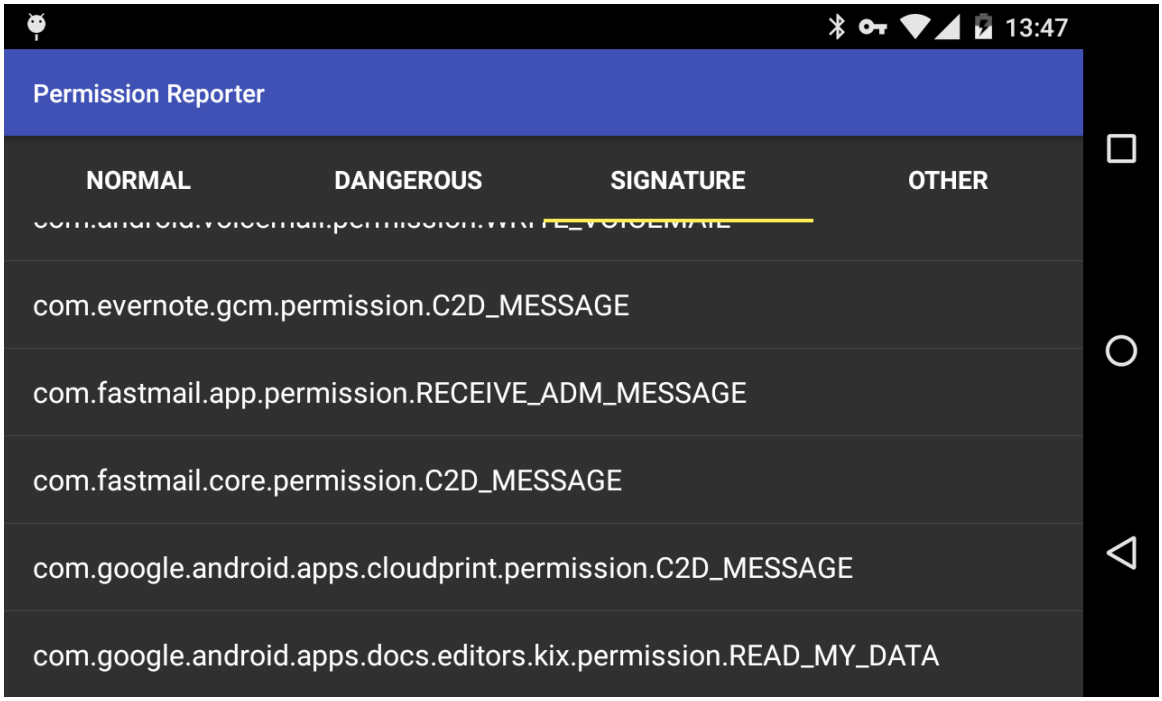


Figure 749: PermissionReporter, Other Tab, on Android 6.0

The roster will include both system permissions, Play Services-defined permissions, and third-party permissions:



Restricted Profiles and UserManager

Android 4.2 introduced the concept of having multiple distinct users of a tablet. Each user would get their own portion of internal and external storage, as if they each had their own tablet.

Android 4.3 extends this a bit further, with the notion of setting up restricted profiles. As the name suggests, a restricted profile is restricted, in terms of what it can do on the device. Some restrictions will be device-wide (e.g., can the user install apps?), and some restrictions will be per-app. You can elect to allow your app to be restricted, where you define the possible ways in which your app can be restricted, and the one setting up the restricted profile can then configure the desired options for some specific profile.

This chapter will explain how users set up these restricted profiles, what you can learn about the device-wide restrictions, and how you can offer your own restrictions for your own app.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on files and its section on multiple user accounts.

Android Tablets and Multiple User Accounts

The theory is that tablets are likely to be shared, whether among family members, among team members in a business, or similar sorts of group settings. There are three levels of "user" in an Android 4.3+ tablet that we will need to consider.

Primary User

The primary user is whoever first set up the tablet after initial purchase. In a family, this is probably a parent; in a corporate setting, this might be an IT administrator.

Prior to Android 4.2, there was only one user per device, and that user could (generally) do anything. In Android 4.2+, the primary user holds this role.

One thing that the primary user can do is set up other users, via the Users option in the Settings app:

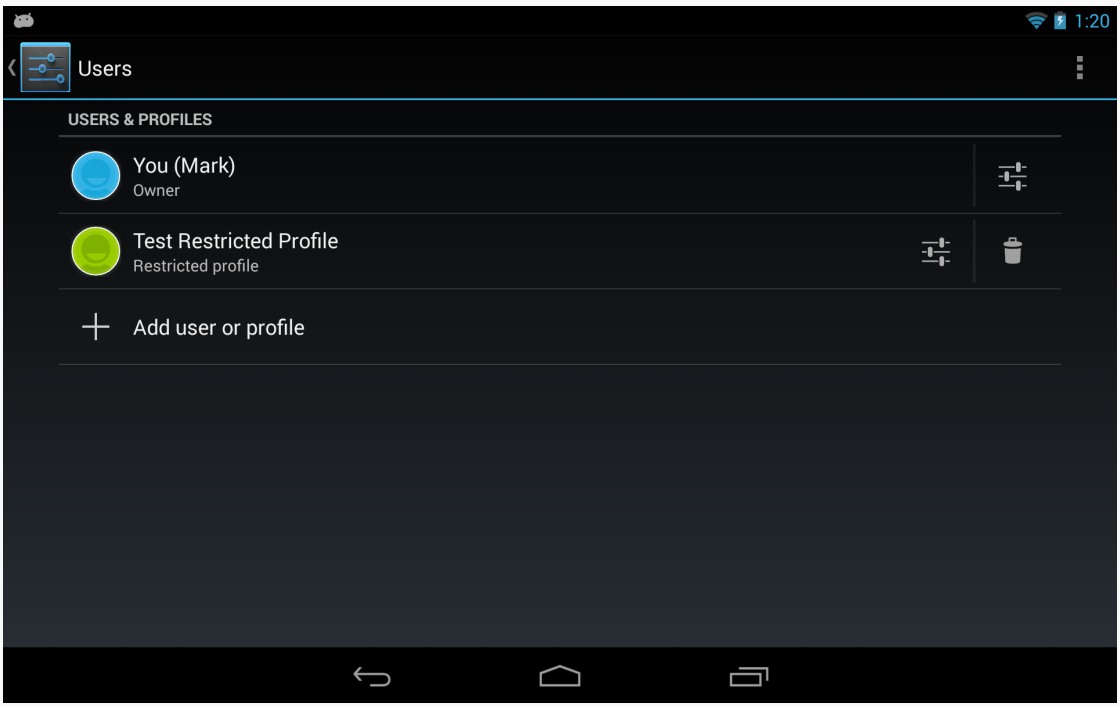


Figure 750: Users Screen in Settings

Tapping the “Add user or profile” entry allows the primary user to set up another user or restricted profile:

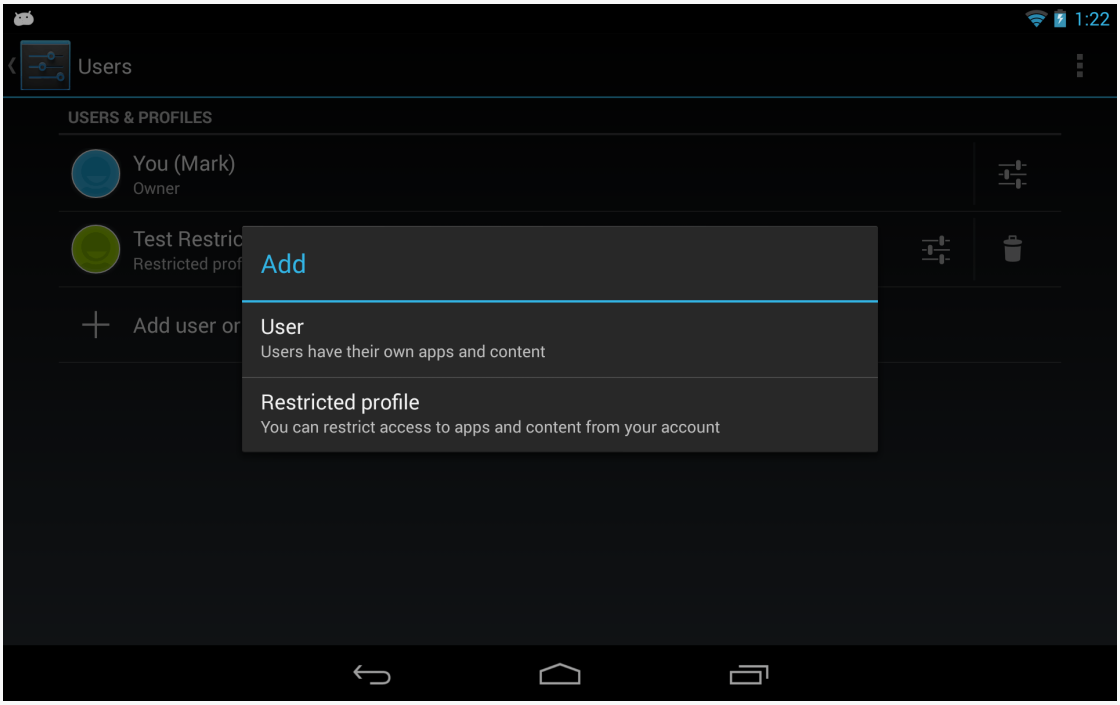


Figure 751: Add Dialog in Users Screen in Settings

Secondary User

Choosing “User” from the Add dialog will define a secondary user of the device. This user has much of the same control as the primary user, in terms of being able to install and run whatever apps are desired.

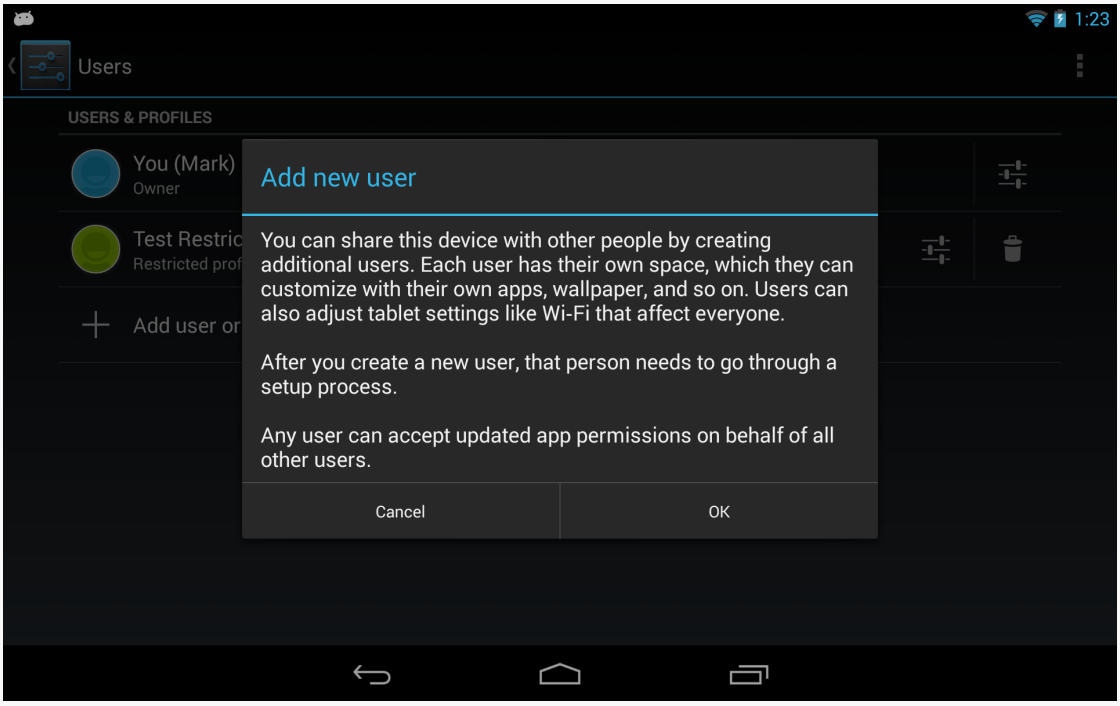


Figure 752: Add New User Warning Dialog in Users Screen in Settings

Restricted Profile

A restricted profile is akin to a secondary user, in that it gets its own separate portion of internal and external storage. Beyond that, though, the primary user can further configure what the restricted profile can access:

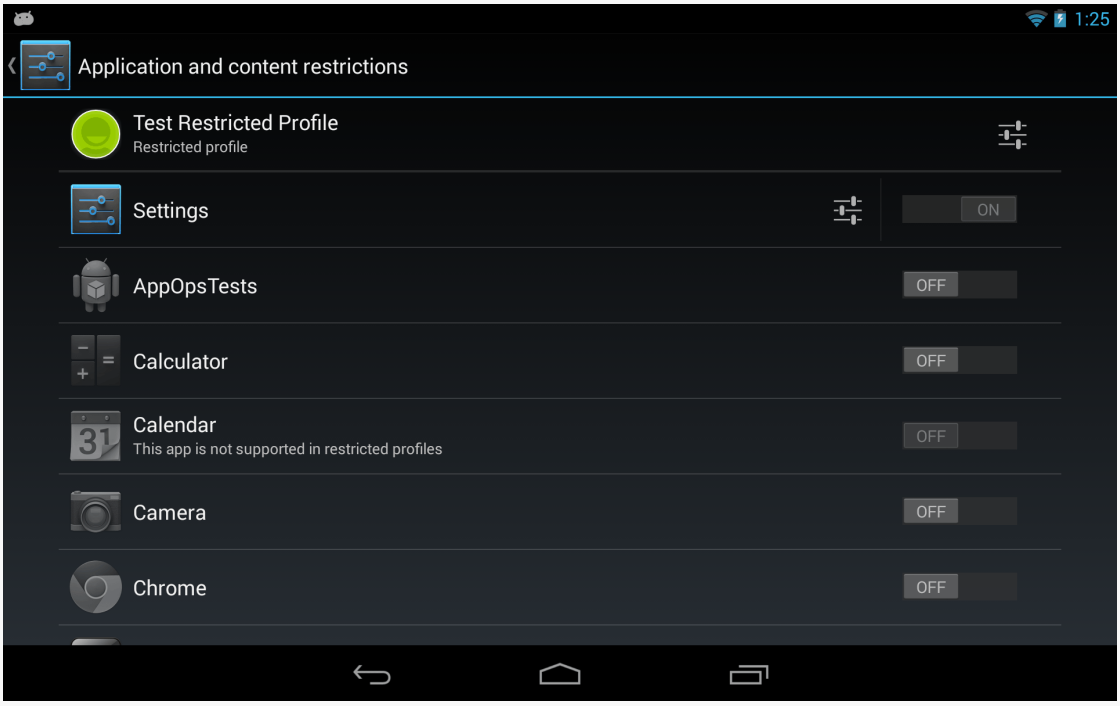


Figure 753: Restricted Profile Configuration Screen in Settings

The bulk of the restricted profile configuration screen is a list of apps, with `Switch` widgets to allow the primary user to allow or deny access to each app.

Some apps will have the "settings" icon to the left of the `Switch`. Tapping that will either bring up a dedicated activity for restricting operations within that app, or it will add new rows to the list with individual restriction options for that app. For example, tapping the settings icon for the Settings app adds a row where the primary user can block location sharing:

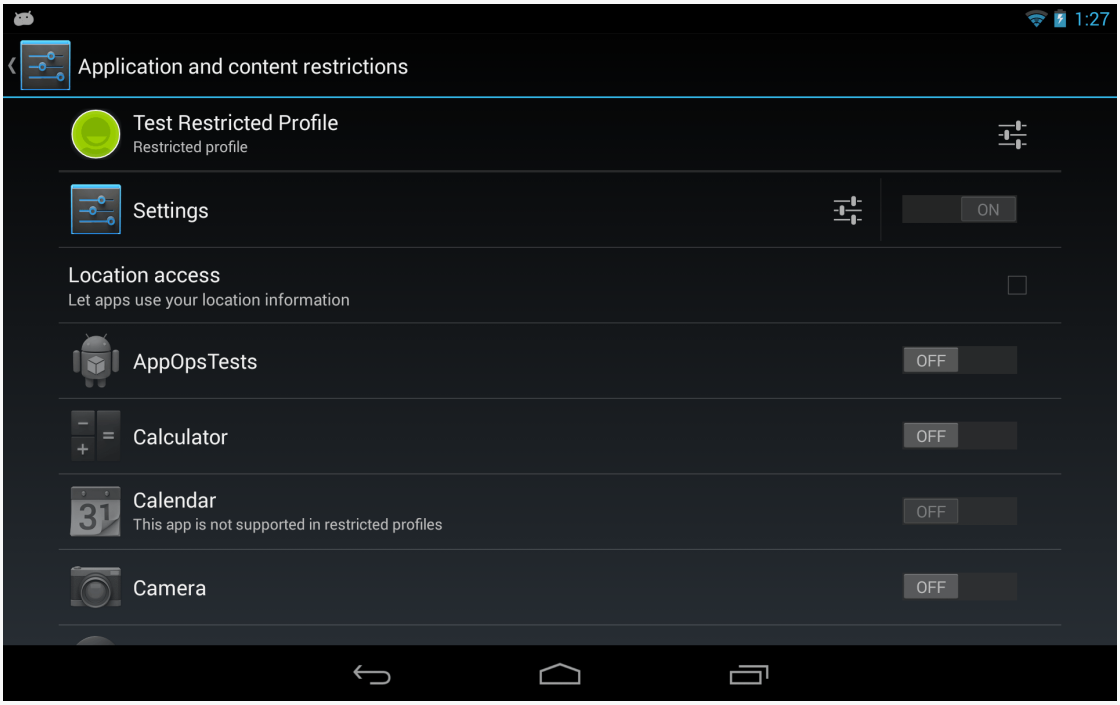


Figure 754: Location Sharing Restrictions

The “settings” icon in the first row, for the profile itself, will allow the primary user to control things for the entire profile, notably its name.

Switching to the restricted profile (e.g., via the lockscreen) will show the constrained set of available apps:

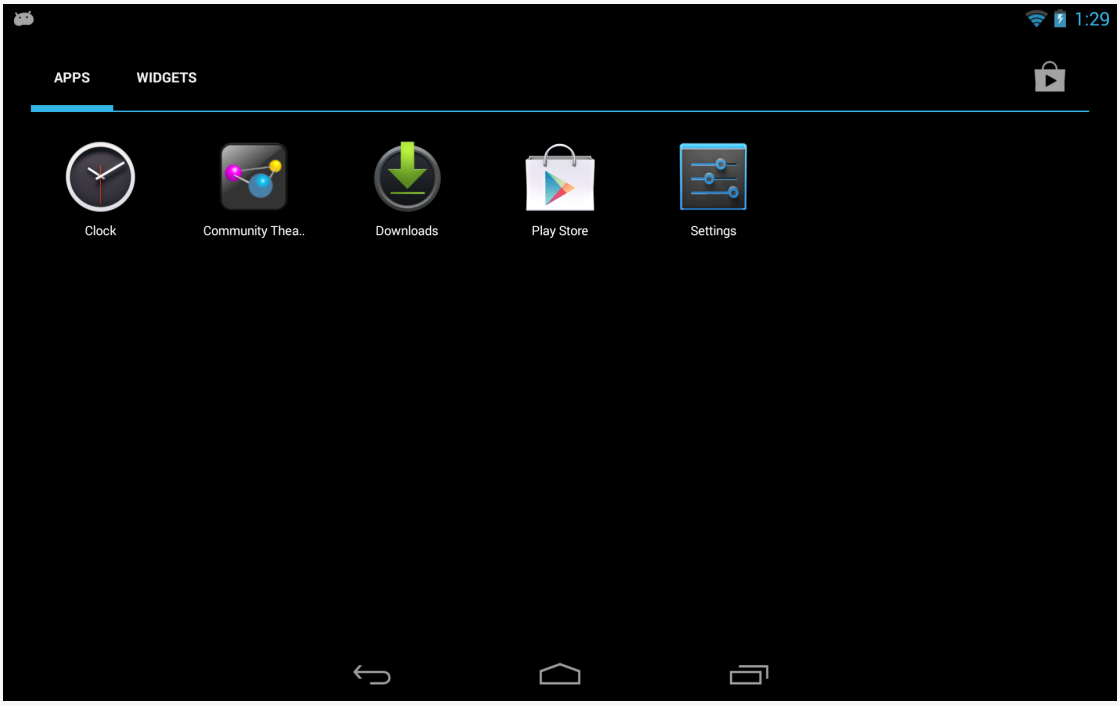


Figure 755: Apps in a Restricted Profile

Determining What the User Can Do

Your app can find out what device-level restrictions were placed on the current user by means of the `UserManager` system service. Specifically, as you can see in `MainActivity` of the [RestrictedProfiles/Device]() sample project, all you need to do is:

- Acquire an instance of a `UserManager` by calling `getSystemService()` on a `Context`, passing in `USER_SERVICE` as the service's name
- Calling `getUserRestrictions()` on the `UserManager`:

```
package com.commonsware.android.profile.device;

import android.app.Activity;
import android.os.Bundle;
import android.os.UserManager;
import android.widget.Toast;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    UserManager mgr=(UserManager)getSystemService(USER_SERVICE);
    Bundle restrictions=mgr.getUserRestrictions();

    if (restrictions.keySet().size() > 0) {
      setContentView(R.layout.activity_main);

      RestrictionsFragment f=
          (RestrictionsFragment)getFragmentManager().findFragmentById(R.id.contents);

      f.showRestrictions(restrictions);
    }
    else {
      Toast.makeText(this, R.string.no_restrictions, Toast.LENGTH_LONG)
           .show();
      finish();
    }
  }
}
```

(from RestrictedProfiles/Device/app/src/main/java/com/commonsware/android/profile/device/MainActivity.java)

`getUserRestrictions()` returns a `Bundle`, whose keys are documented on `UserManager` for various device-level restrictions that theoretically can be placed on the user. Here, "theoretically" means that while `UserManager` documents several `DISALLOW_*` constants, only two seem to be directly accessible to the primary user for configuration via Settings:

- `DISALLOW_MODIFY_ACCOUNTS`, to prevent a restricted profile from, among other things, modifying restricted profiles
- `DISALLOW_SHARE_LOCATION`, to prevent the apps run in this restricted profile from gathering location data

`MainActivity` examines the `Bundle` and, if it is empty, just displays a `Toast` and exits via `finish()`. This is the behavior you will see if you run this sample app on a non-restricted profile, such as the primary user. If, however, the `Bundle` has one or more keys, we inflate an `activity_main` layout that contains a `RestrictionsFragment` in a `<fragment>` element:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/contents"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  class="com.commonsware.android.profile.device.RestrictionsFragment"/>
```

(from RestrictedProfiles/Device/app/src/main/res/layout/activity_main.xml)

We then retrieve the `RestrictionsFragment` from the `FragmentManager` and call `showRestrictions()` on it, passing in the `Bundle`.

`RestrictionsFragment` is a `ListFragment` employing a custom `RestrictionsAdapter`. The `RestrictionsAdapter` wraps around the `Bundle` and an `ArrayList` of its keys. The `RestrictionsAdapter` constructor creates the `ArrayList` by sorting the `keySet()` of the `Bundle`. `getView()` on `RestrictionsAdapter` lets the superclass handle inflating the row (`android.R.layout.simple_list_item_1`), then puts an icon on the right side by using `setCompoundDrawablesWithIntrinsicBounds()`, which can tuck a drawable resource onto any of the four sides of a `TextView`.

The resulting list will show green icons for keys where the `Bundle` has stored a `true` `Boolean` value, and a red icon for `false`:

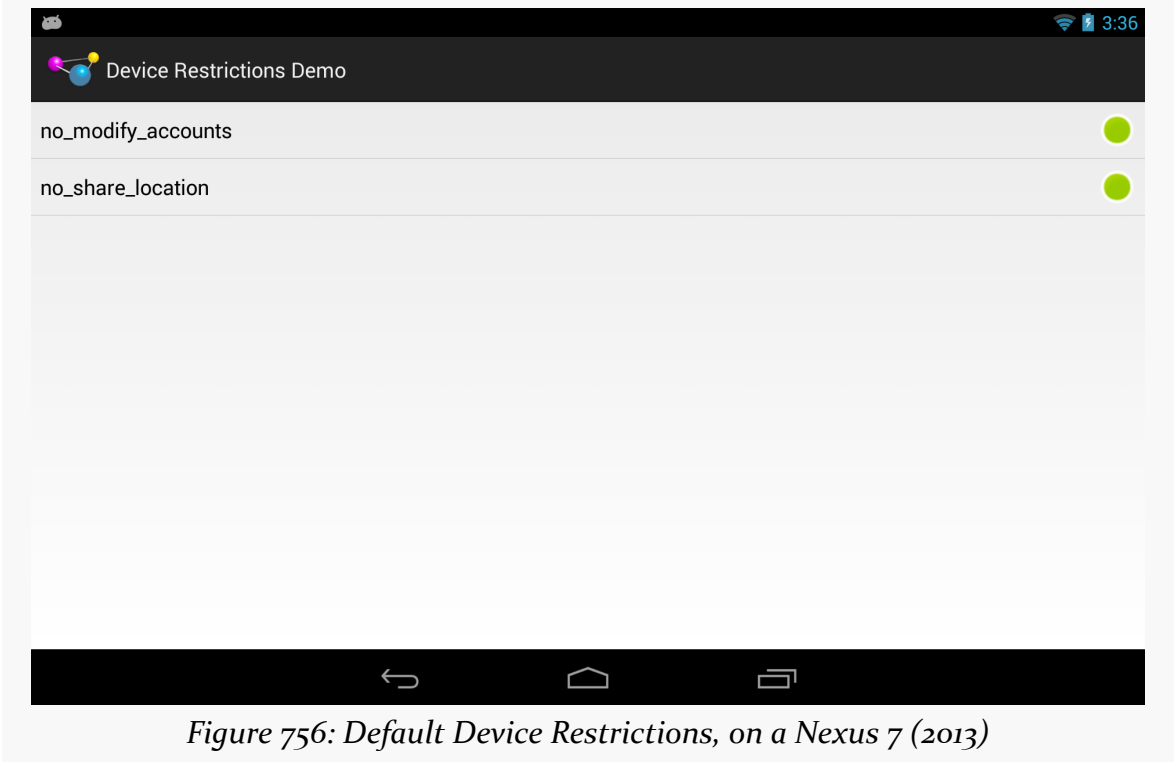


Figure 756: Default Device Restrictions, on a Nexus 7 (2013)

Since the keys are negative in tone (e.g., `DISALLOW_MODIFY_ACCOUNTS`), `true` means that the restriction is enforced and the underlying operation (e.g., modifying accounts) *cannot* be done.

Impacts of Device-Level Restrictions

Your app's functionality may be limited by these device-level restrictions. This section outlines some of the results you should expect from a restricted profile.

Restricting Location Access

If a restricted profile is prevented from sharing the device's location with apps, those apps simply will not receive location updates. There is no good way to detect this via the location API (e.g., `isProviderEnabled()` returns `true`), so you will have to detect this via `getUserRestrictions()` on `UserManager` as noted above.

Uninstalling Apps

Even without specific configuration, the restricted profile can only uninstall apps that they are available to that profile. However, since apps are really shared between profiles, this only removes that app from the restricted profile; it does not actually uninstall the app from the device as a whole.

Enabling Custom Restrictions

As noted earlier, the list of apps that is shown on the restricted profile configuration screen in Settings can have "settings" icons. The Settings app itself will have a settings icon, to allow the primary user to configure device-level restrictions.

But, what if you want *your* app to have such a settings icon? Maybe it makes sense for your app to allow the primary user to restrain restricted profiles from doing certain things within your app:

- Block in-app purchases
- Only show certain categories of content, not the full roster
- Only allow operation during certain times of the day

The means by which the Settings app restricts profiles is also available to you. You can declare to Android what aspects of your app can be restricted. Android will then collect that restriction data for you. Your app, at runtime, can then determine what restrictions are in place (if any) and take appropriate steps.

All of this will be illustrated using the `RestrictedProfiles/App` sample project.

Stating Your Restrictions

The biggest thing that you need to do to restrict your app is teach Android how to collect restrictions. In other words, you need to tell Android what to do when the user taps that settings icon in the restricted profile entry for your app.

You have two major options:

- Provide a list of the restrictions that Android should render and collect itself, or
- Provide an `Intent` that can be used to start up an activity of your own design where you collect those restrictions

Either approach will require you to set up a manifest-registered `BroadcastReceiver`, set to respond to the `android.intent.action.GET_RESTRICTION_ENTRIES` action:

```
    <receiver android:name="RestrictionEntriesReceiver">
      <intent-filter>
        <action android:name="android.intent.action.GET_RESTRICTION_ENTRIES"/>
      </intent-filter>
    </receiver>
```

(from RestrictedProfiles/App/app/src/main/AndroidManifest.xml)

That `BroadcastReceiver` will be called with `sendOrderedBroadcast()`, not so much to affect ordering, but to allow the `BroadcastReceiver` to send back a result via its `setResultExtras()` method. This provides a `Bundle` that the broadcaster can eventually retrieve, in this case providing details of what restrictions we wish to collect from the primary user to restrict the profile.

Option #1: RestrictionEntry List

To collect restrictions the way the Settings app does — with restriction rows appearing below your app in the restricted profile screen in Settings – your `BroadcastReceiver` will need to put an entry into the return `Bundle`, under the key of `EXTRA_RESTRICTIONS_LIST` (a constant defined on the `Intent` class). The value needs to be an `ArrayList` of `RestrictionEntry` objects, with each `RestrictionEntry` describing one restriction to collect.

Another thing that the `RestrictionEntry` objects contain is their current value. Android itself retains these values and supplies them to your `BroadcastReceiver` via an `EXTRA_RESTRICTIONS_BUNDLE` extra on the incoming `Intent`. Your app needs to

use those current values when constructing its list of `RestrictionEntry` objects to return.

So, let's take a look at `RestrictionEntriesReceiver`, the receiver we have set up to handle the `android.intent.action.GET_RESTRICTION_ENTRIES` action for this sample app.

The entry point for `RestrictionEntriesReceiver` is `onReceive()`, as it is for any basic `BroadcastReceiver`:

```
@Override
public void onReceive(Context ctxt, Intent intent) {
  Bundle current=
      (Bundle)intent.getParcelableExtra(Intent.EXTRA_RESTRICTIONS_BUNDLE);
  ArrayList<RestrictionEntry> restrictions=
      new ArrayList<RestrictionEntry>();

  restrictions.add(buildBooleanRestriction(ctxt, current));
  restrictions.add(buildChoiceRestriction(ctxt, current));
  restrictions.add(buildMultiSelectRestriction(ctxt, current));

  Bundle result=new Bundle();

  result.putParcelableArrayList(Intent.EXTRA_RESTRICTIONS_LIST,
                                restrictions);

  setResultExtras(result);
}
```

(from RestrictedProfiles/App/app/src/main/java/com/commonsware/android/profile/app/RestrictionEntriesReceiver.java)

In `onReceive()`, `RestrictionEntriesReceiver` pulls out the `Bundle` of current restrictions, by retrieving the `EXTRA_RESTRICTIONS_BUNDLE` extra from the `Intent` passed into `onReceive()`. Note that this `Bundle` could very well be empty, if this is the first time we are being asked for restrictions.

`RestrictionEntriesReceiver` creates an empty `ArrayList` of `RestrictionEntry` objects, then calls a series of builder methods to create a total of three such `RestrictionEntry` objects, adding each to the list. `onReceive()` goes on to create a `Bundle` representing the results to be returned, packages the `ArrayList` in that `Bundle` under the `EXTRA_RESTRICTIONS_LIST` key, and returns that `Bundle` to the caller by means of `setResultExtras()`.

The three builder methods are each responsible for defining a single `RestrictionEntry`, including populating it with the current value from the `current Bundle`.

There are three types of `RestrictionEntry`, for boolean, single-selection lists ("choice"), and multi-selection lists. The `RestrictionEntry` constructor takes two parameters:

- The `String` key under which we will later retrieve this restriction value
- The current value of the restriction

The current value is:

- A `boolean` for boolean restrictions
- A `String` for choice restrictions
- A `String` array for multi-select restrictions

Our first builder, `buildBooleanRestriction()`, populates and returns a `RestrictionEntry` designed to collect a boolean value from the primary user, via a `CheckBox`:

```
private RestrictionEntry buildBooleanRestriction(Context ctxt,
                                                 Bundle current) {
  RestrictionEntry entry=
      new RestrictionEntry(RESTRICTION_BOOLEAN,
                           current.getBoolean(RESTRICTION_BOOLEAN,
                                              false));

  entry.setTitle(ctxt.getString(R.string.boolean_restriction_title));
  entry.setDescription(ctxt.getString(R.string.boolean_restriction_desc));

  return(entry);
}
```

(from RestrictedProfiles/App/app/src/main/java/com/commonsware/android/profile/app/RestrictionEntriesReceiver.java)

`buildBooleanRestriction()` retrieves the current value from `current Bundle` to use with the `RestrictionEntry` constructor. In this case, if there is no such entry in the `Bundle`, the overall default value is `false`.

Each `RestrictionEntry` can have a title (`setTitle()`), supplying a string which will be displayed to describe what this restriction is. A boolean restriction can also have a description (`setDescription()`), containing another string with a bit more text.

Note that, at the present time, the other two types of restrictions will ignore any description that you include. Also note that the values supplied to `setTitle()` and `setDescription()` need to be strings, and so if you wish to use a string resource, you will need to get the actual string value yourself via `getString()`.

The remaining two builder methods have a similar structure:

```
private RestrictionEntry buildChoiceRestriction(Context ctxt,
                                                Bundle current) {
  RestrictionEntry entry=
      new RestrictionEntry(RESTRICTION_CHOICE,
                           current.getString(RESTRICTION_CHOICE));

  entry.setTitle(ctxt.getString(R.string.choice_restriction_title));
  entry.setChoiceEntries(ctxt, R.array.display_values);
  entry.setChoiceValues(ctxt, R.array.restriction_values);

  return(entry);
}

private RestrictionEntry buildMultiSelectRestriction(Context ctxt,
                                                     Bundle current) {
  RestrictionEntry entry=
      new RestrictionEntry(RESTRICTION_MULTI,
                           current.getStringArray(RESTRICTION_MULTI));

  entry.setTitle("A Multi-Select Restriction");
  entry.setChoiceEntries(ctxt, R.array.display_values);
  entry.setChoiceValues(ctxt, R.array.restriction_values);

  return(entry);
}
```

(from RestrictedProfiles/App/app/src/main/java/com/commonsware/android/profile/app/RestrictionEntriesReceiver.java)

As with a `ListPreference`, you provide two string arrays to the `RestrictionEntry`, representing the values the primary user sees (`setChoiceEntries()`) and the corresponding values to be supplied to your app based upon the choice(s) (`setChoiceValues()`). You can supply these either as Java string arrays or as `<string-array>` resources – `RestrictionEntriesReceiver` goes with the latter approach.

Option #2: Custom Restriction Activity

It may be that what you want to collect, in terms of restrictions, cannot readily be represented in the form of `Switch` widgets and list dialogs. For example, to restrict use of your app based on time, it would be nice to use a `TimePickerDialog` or the equivalent.

The alternative to returning an `EXTRA_RESTRICTIONS_LIST` roster of `RestrictionEntry` objects from your `BroadcastReceiver` is to have the result `Bundle` contain `EXTRA_RESTRICTIONS_INTENT`. This key should point to an `Intent` that identifies the activity that you want to start up when the user taps the settings icon. Android will call `startActivityForResult()` on that `Intent` when the user taps on the settings icon.

Your job is to collect the restrictions from the user, using the `EXTRA_RESTRICTIONS_BUNDLE` from the incoming `Intent` to pre-populate your activity, if desired. When the user is done, you should call `setResult()`, passing in an `Intent` that contains another `EXTRA_RESTRICTIONS_BUNDLE` with the revised data, or optionally a `EXTRA_RESTRICTIONS_LIST` (with the `RestrictionEntry` objects containing the values to be used).

What the Primary User Sees

Given the `RestrictionEntriesReceiver` described above, when the primary user goes to configure a restricted profile, your app will appear with a settings icon next to it:

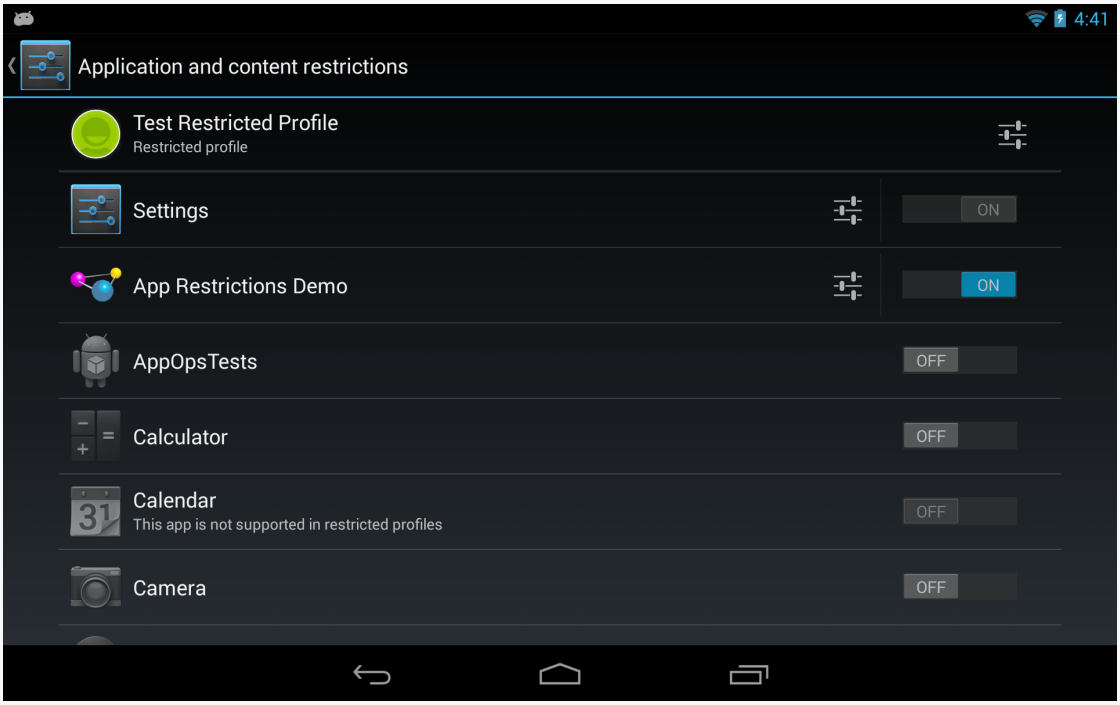


Figure 757: Restricted Profile, Showing App Settings Icon

Tapping that settings icon will "unfold" and display the restrictions that you configured via the `RestrictionEntry` objects:

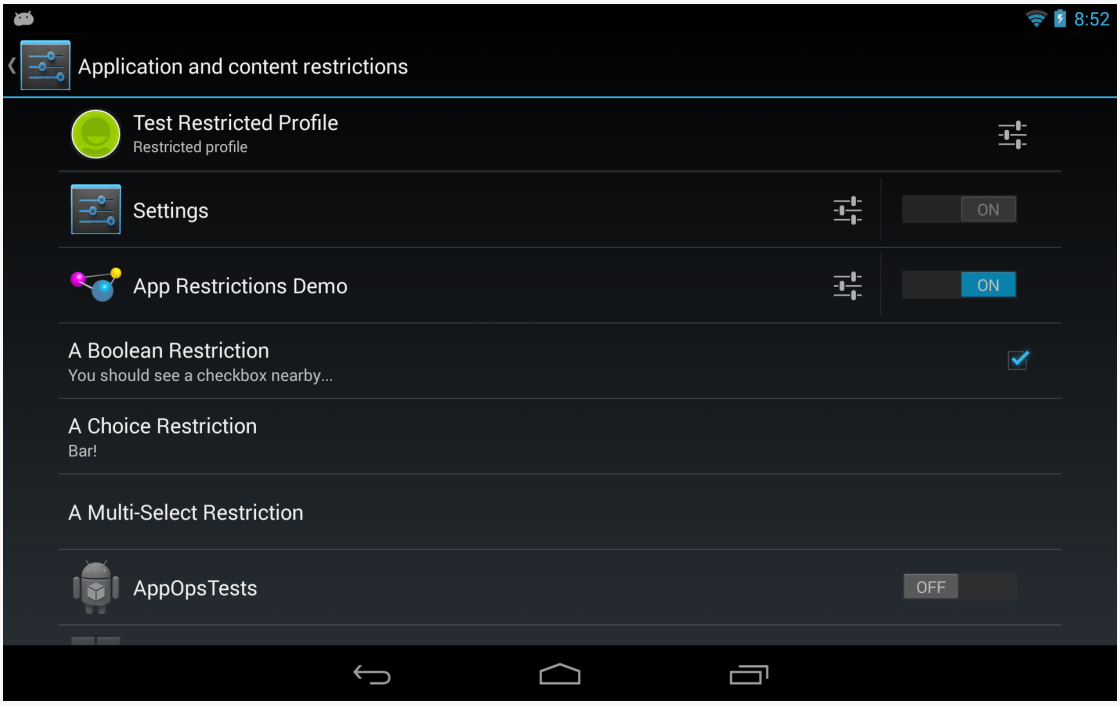


Figure 758: Restricted Profile, Showing App Restrictions

The primary user can then interact with your restrictions, toggling checkboxes and popping up the list dialogs:

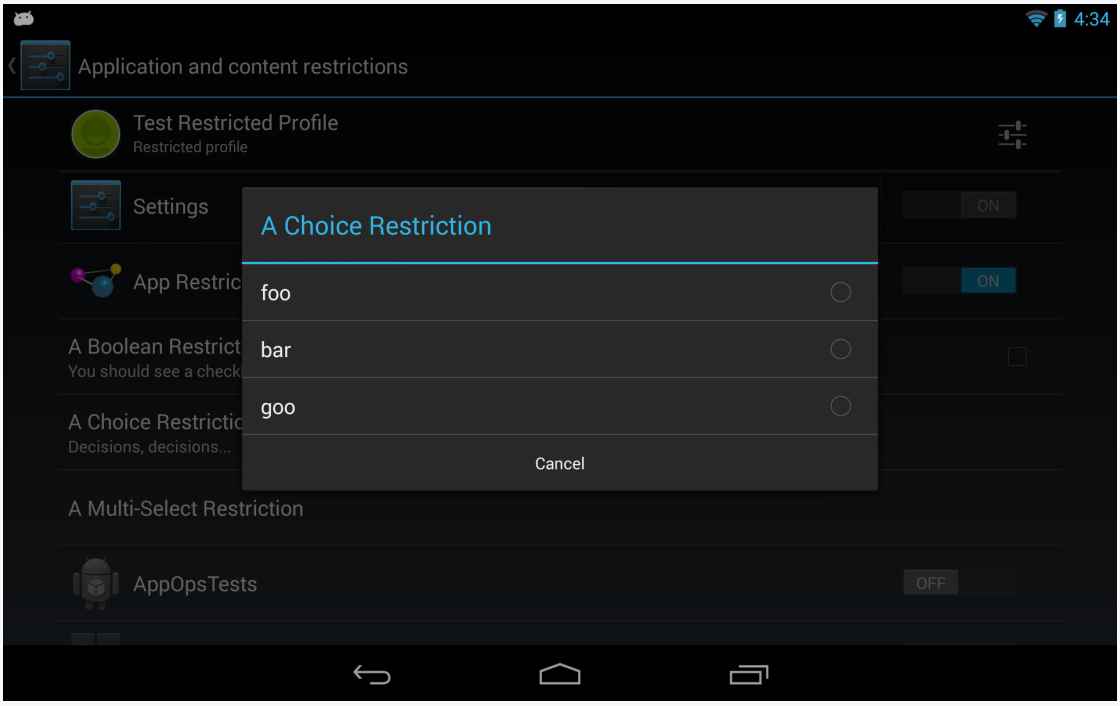


Figure 759: Restricted Profile, Showing Choice Restriction

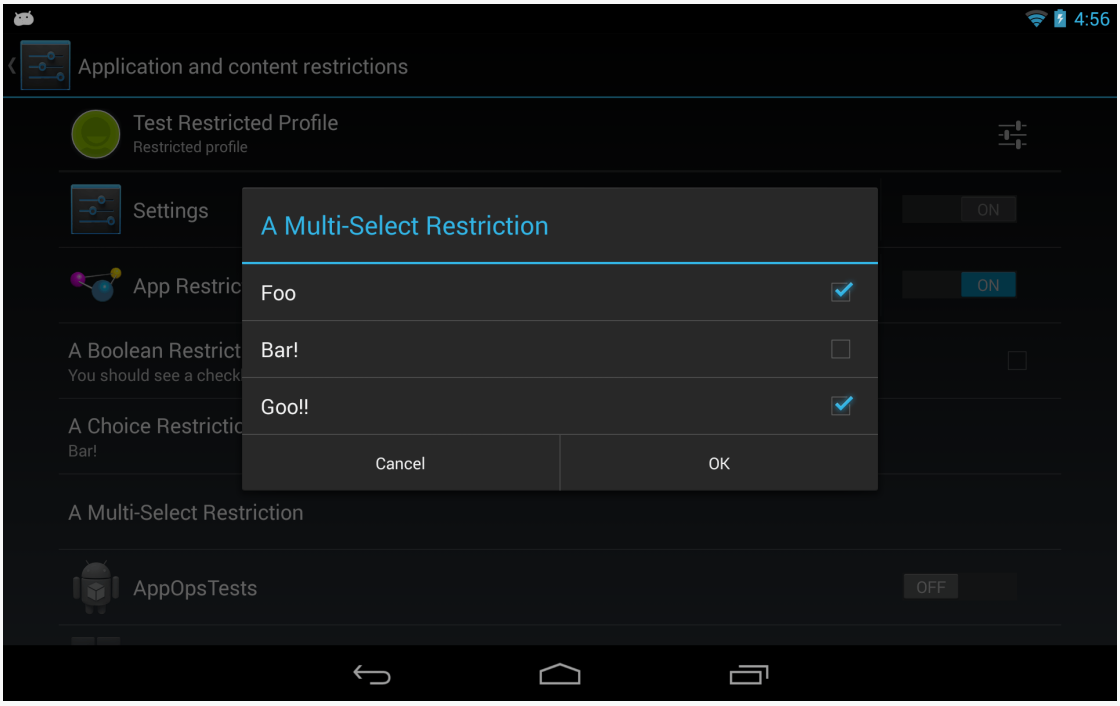


Figure 760: Restricted Profile, Showing Multi-Select Restriction

Finding Out the Current Restrictions

Now, the rest of your app needs to find out what restrictions are placed upon it, so behavior can be tailored accordingly. To do this, call `getApplicationRestrictions()` on `UserManager`, passing in your package name, as seen here in `MainActivity`:

```
package com.commonsware.android.profile.app;

import android.app.Activity;
import android.os.Bundle;
import android.os.UserManager;
import android.widget.Toast;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    UserManager mgr=(UserManager)getSystemService(USER_SERVICE);
    Bundle restrictions=
        mgr.getApplicationRestrictions(getPackageName());

    if (restrictions.keySet().size() > 0) {
      setContentView(R.layout.activity_main);

      RestrictionsFragment f=
          (RestrictionsFragment)getFragmentManager().findFragmentById(R.id.contents);

      f.showRestrictions(restrictions);
    }
    else {
      Toast.makeText(this, R.string.no_restrictions, Toast.LENGTH_LONG)
           .show();
      finish();
    }
  }
}
```

(from RestrictedProfiles/App/app/src/main/java/com/commonsware/android/profile/app/MainActivity.java)

This `Bundle` could be empty, or it could have values specified by the primary user to restrict the profile that is running your app.

In the case of this sample, we once again set up a `RestrictionsAdapter` to show the results, if the `Bundle` is not empty. However, our adapter is a bit more complicated, as there are more than boolean restrictions now. `getView()` has been updated to handle all three possible restrictions, showing the icon for the boolean restriction, and showing the value(s) from the lists in the other restrictions:

```
package com.commonsware.android.profile.app;

import android.app.ListFragment;
```

```
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.Collections;

public class RestrictionsFragment extends ListFragment {
  public void showRestrictions(Bundle restrictions) {
    setListAdapter(new RestrictionsAdapter(restrictions));
  }

  class RestrictionsAdapter extends ArrayAdapter<String> {
    Bundle restrictions;

    RestrictionsAdapter(Bundle restrictions) {
      super(getActivity(), android.R.layout.simple_list_item_1,
            new ArrayList<String>());

      ArrayList<String> keys=
          new ArrayList<String>(restrictions.keySet());

      Collections.sort(keys);
      addAll(keys);

      this.restrictions=restrictions;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      TextView row=
          (TextView)super.getView(position, convertView, parent);
      String key=getItem(position);

      if (RestrictionEntriesReceiver.RESTRICTION_BOOLEAN.equals(key)) {
        int icon=
            restrictions.getBoolean(key) ? R.drawable.ic_true
                : R.drawable.ic_false;

        row.setCompoundDrawablesWithIntrinsicBounds(0, 0, icon, 0);
      }
      else if (RestrictionEntriesReceiver.RESTRICTION_CHOICE.equals(key)) {
        row.setText(String.format("%s (%s)", key,
                                  restrictions.getString(key)));
      }
      else {
```

```
        String value=
            TextUtils.join(" | ", restrictions.getStringArray(key));

        row.setText(String.format("%s (%s)", key, value));
      }

      return(row);
    }
  }
}
```

(from RestrictedProfiles/App/app/src/main/java/com/commonsware/android/profile/app/RestrictionsFragment.java)

The result, when run on a restricted profile with restrictions placed upon our app, is to show those restrictions:

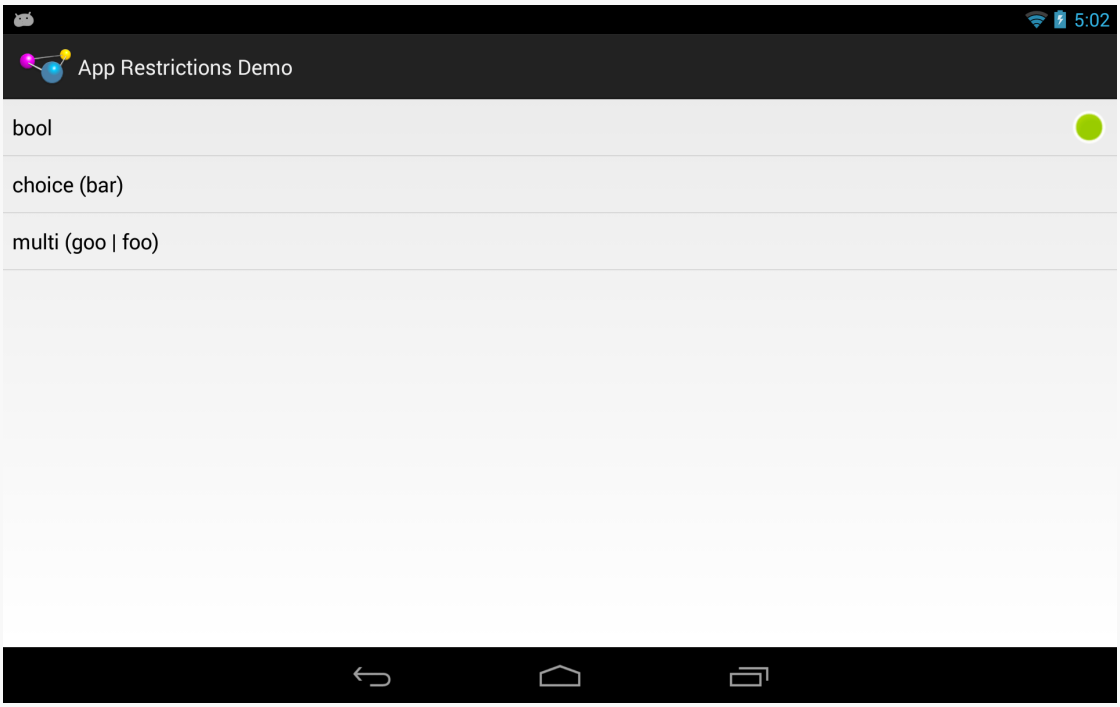


Figure 761: App Restrictions Demo, on a Restricted Profile

Implicit Intents May Go “Boom”

The primary user of a tablet, when setting up a restricted profile, can control what apps are available to that profile. In many cases, if the user is setting up a restricted

profile in the first place, the list of apps available to that profile will be fairly limited, such as only allowing a young child to access a few games and educational apps.

`startActivity()` always has the chance of throwing an `ActivityNotFoundException`. However, for certain `Intent` actions, we often ignore this possibility, because we are certain that there will be an app that can handle our request:

- All Android devices have Web browsers, right?
- All Android devices have some sort of mapping application, right?
- All Android devices let you pick a contact, right?

Now, with restricted profiles, you will need to deal with the `ActivityNotFoundException` case all of the time. You have three basic approaches for this:

1. Wrap all `startActivity()` and `startActivityForResult()` calls in a `try/catch` block that catches `ActivityNotFoundException` and intelligently handle the problem
2. Use `PackageManager` and `resolveActivity()` *before* trying to start the activity, where if `resolveActivity()` returns `null`, you know that there is no activity available to handle your desired operation
3. Switch out some of your `startActivity()` and `startActivityForResult()` calls for implementations in your app (e.g., embed Maps V2 rather than try to launch a potentially-nonexistent activity)

You might consider implementing a `safeStartActivity()` utility method that wraps up your particular plan, so you can debug it once.

Device Authentication

Android devices have supported device authentication since the beginning. Users could lock their devices with a PIN, passphrase, or "pattern", to help deter unauthorize use. Nowadays, fingerprint sensors are available on some Android devices as another authentication option.

For the vast majority of apps, none of this matters.

Some apps have sensitive content (e.g., password manager) or offer sensitive services (e.g., banking app). These apps may want to tie more tightly into the device authentication options, to ensure that the person working with the app right now is the device's authorized user.

In this chapter, we will look at how to perform this sort of integration.

Prerequisites

One of the examples in this chapter makes use of RxJava.

Is the Device Secure?

You cannot validate the identity of the user on a device that is not secured by something (PIN, passphrase, pattern, fingerprint, etc.). So, one of the first things you may wish to determine is whether or not such a lock has been applied to the device.

Android has two separate ways of accomplishing this, with slightly different feature sets.

The DevicePolicyManager Approach

In the chapter on device administration, we cover a number of APIs for managing the lock quality. In particular, `DevicePolicyManager` has the concept of "password quality". Any app — not just designated device admin apps — can call `getPasswordQuality()` on a `DevicePolicyManager` instance to find out the designated minimum required quality. A value of `PASSWORD_QUALITY_UNSPECIFIED` indicates that "anything goes". `PASSWORD_QUALITY_SOMETHING` indicates that the device needs some form of authentication, but does not establish any criteria beyond that. Other values start to put restrictions on the authentication option, such as `PASSWORD_QUALITY_ALPHABETIC`, indicating that the authentication cannot be in the form of a simple PIN.

Other methods let you get minimum required password lengths, minimum required characters of certain classes (e.g., symbols), and so forth.

It is fairly likely that the current authentication option in force adheres to the password criteria exposed by those methods.

The KeyguardManager Approach

The `KeyguardManager` system service has a simpler API... with a couple of wrinkles.

- `isDeviceSecure()` returns `true` if the device is secured with a PIN, pattern, or password. However, this method is new to API Level 23.
- `isKeyguardSecure()` returns `true` if the device is secured with a PIN, pattern, password, or SIM card lock. A "SIM card lock" amounts to a PIN associated with the SIM card, one that follows the SIM card around if you move it between devices. Conversely, a device that is secured solely by a SIM card lock can be accessed simply by removing the SIM card. This method is available on API Level 16 and higher devices.

Since SIM card locks do not seem very common, the two methods will tend to return the same result. However, since SIM card lock security is fairly weak, on Android 6.0+, use `isDeviceSecure()`, so SIM card locks are not taken into account with the result.

The `DeviceAuth/SecureCheck` sample application demonstrates the use of these methods (and of a few others that we will see shortly).

Our activity's UI consists of two big icons with those two `KeyguardManager` method names as captions.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="com.commonsware.android.auth.check.MainActivity">

  <TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:text="@string/label_device"
    android:textAlignment="viewStart"
    android:textAppearance="?android:textAppearanceLarge"
    android:typeface="monospace"
    app:layout_constraintBottom_toBottomOf="@id/device"
    app:layout_constraintLeft_toRightOf="@id/device"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="@id/device" />

  <ImageView
    android:id="@+id/device"
    android:layout_width="64dp"
    android:layout_height="64dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:text="@string/label_keyguard"
    android:textAlignment="viewStart"
    android:textAppearance="?android:textAppearanceLarge"
    android:typeface="monospace"
    app:layout_constraintBottom_toBottomOf="@id/keyguard"
    app:layout_constraintLeft_toRightOf="@id/keyguard"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="@id/keyguard" />

  <ImageView
    android:id="@+id/keyguard"
    android:layout_width="64dp"
    android:layout_height="64dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/device" />

</android.support.constraint.ConstraintLayout>
```

(from DeviceAuth/SecureCheck/app/src/main/res/layout/activity_main.xml)

In `onCreate()`, we retrieve a `KeyguardManager` system service and set the images on those icons based on the `KeyguardManager` results:

- If the method returns `true`, we show a locked icon
- If the method returns `false`, we show an unlocked icon
- If the method cannot be called (`isDeviceSecure()` on pre-Android 6.0 devices), we show a question mark icon, indicating that we do not know the result

```
private KeyguardManager mgr;

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  mgr=(KeyguardManager)getSystemService(KEYGUARD_SERVICE);

  ImageView device=findViewById(R.id.device);
  ImageView keyguard=findViewById(R.id.keyguard);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
    if (mgr.isDeviceSecure()) {
      device.setImageResource(R.drawable.ic_lock_black_24dp);
    }
    else {
      device.setImageResource(R.drawable.ic_lock_open_black_24dp);
    }
  }
  else {
    device.setImageResource(R.drawable.ic_help_black_24dp);
  }

  if (mgr.isKeyguardSecure()) {
    keyguard.setImageResource(R.drawable.ic_lock_black_24dp);
  }
  else {
    keyguard.setImageResource(R.drawable.ic_lock_open_black_24dp);
  }
}
```

(from DeviceAuth/SecureCheck/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

Running the app will show the results of those calls:

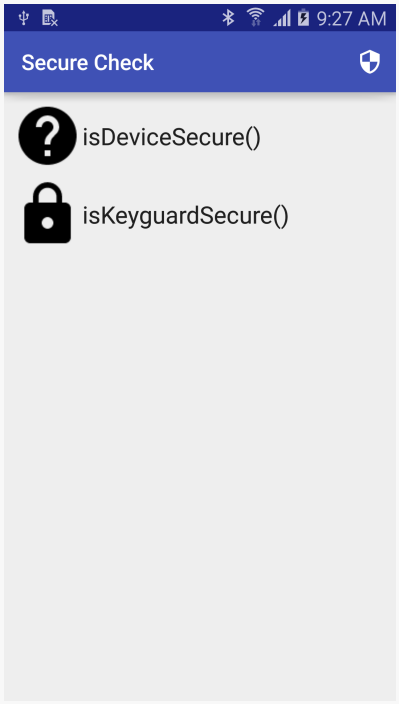


Figure 762: Secure Check, As Run On Secure Android 5.0 Device

Figure 763: Secure Check, As Run On Insecure Android 6.0 Device

Reconfirming the User

Those methods are interesting, but they do not tell you much about the current user. Obviously, if the device is not secure, you have no idea who the current user is. However, even if the device has a PIN/password/pattern lock, it is possible that the real user unlocked device, but somebody else now has that unlocked device. Some apps will want a way to re-authenticate the user, to confirm that the person working with the device right now is the real user, not somebody else who has their hands on the unlocked device.

`KeyguardManager` has a `createConfirmDeviceCredentialIntent()` method that helps with this. This method returns an `Intent` that can be used with `startActivityForResult()`. What then appears is a system-supplied activity that requires the user to re-authenticate the device. In `onActivityResult()`, a `RESULT_OK` result means that the user successfully re-authenticated. Anything else means that the user failed to re-authenticate and eventually pressed BACK to return to your app.

The `SecureCheck` sample app has an action bar icon that, when tapped, invokes an `authenticate()` method, one that uses `createConfirmDeviceCredentialIntent()`:

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
private void authenticate() {
  Intent i=mgr.createConfirmDeviceCredentialIntent("title", "description");

  if (i==null) {
    Toast.makeText(this, "No authentication required!", Toast.LENGTH_SHORT).show();
  }
  else {
    startActivityForResult(i, REQUEST_CODE);
  }
}
```

(from DeviceAuth/SecureCheck/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

`createConfirmDeviceCredentialIntent()` takes two parameters: a title and a description. In principle, these should appear on the authentication activity, to provide context to the user as to why this screen appeared. In reality, the description may or may not appear, so do not assume that the user will be able to see whatever you put there.

Since this `Intent` is designed for use with `startActivityForResult()`, we cannot directly put it into a `PendingIntent` or anything. If for some reason you needed to re-authenticate from a `Notification` or app widget or something, you can use a `Theme.Translucent.NoTitleBar`-themed activity for the `PendingIntent`, as such

activities are effectively invisible. That activity, in turn, can call `startActivityForResult()` with the `Intent` from `createConfirmDeviceCredentialIntent()`.

Here, we just call `startActivityForResult()` directly, showing a `Toast` in `onActivityResult()` based upon the user's response:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==REQUEST_CODE) {
    if (resultCode==RESULT_OK) {
      Toast.makeText(this, "Authenticated!", Toast.LENGTH_SHORT).show();
    }
    else {
      Toast.makeText(this, "WE ARE UNDER ATTACK!", Toast.LENGTH_SHORT).show();
    }
  }
}
```

(from DeviceAuth/SecureCheck/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

On some devices (e.g., Nexus 5X running Android 8.1), the authentication activity uses `FLAG_SECURE` and cannot be captured in a screenshot. Other devices have no such restriction:

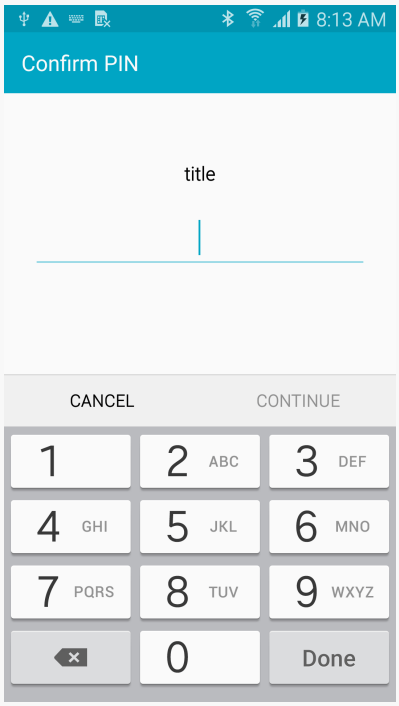


Figure 764: Re-Authentication Activity on a Samsung Galaxy Note 3

Note that only the title appears here; on the Nexus 5X with Android 8.1, both the title and the description appear.

`createConfirmDeviceCredentialIntent()` has a significant limitation: the `Intent` that you create always triggers authentication when started. Sometimes, this is a feature, if you absolutely want to ensure that the user is authenticated before proceeding. Sometimes, this is a bug, if your flow keeps forcing the user to re-authenticate. There is a way to use `createConfirmDeviceCredentialIntent()` to have an authentication timeout, so a successful authentication skips future authentication for some amount of time. This is a bit complex to set up, as it requires custom use of the keystore, and so we will explore it in greater detail in the next chapter.

Fingerprints

Android 6.0 added APIs for fingerprint sensors. For most users, and for most apps, fingerprints are just another aspect of unlocking a device. However, apps can more deeply integrate with the fingerprint sensor if desired.

Note that the libraries and classes related to fingerprints are only supported on API Level 23 and higher.

Authenticating the User

One obvious use of the fingerprint sensor is to authenticate the current user, ensuring that the person holding the device is who we think it is.

There are a few options for doing this, with varying levels of complexity.

Via KeyguardManager

The simplest solution is to not worry about fingerprints, but instead to use the techniques shown earlier in this chapter to authenticate the user.

When a user sets up a fingerprint to use for unlocking a device, that is an adjunct to a more traditional PIN or passphrase. When you start the activity identified by the `Intent` returned by `createConfirmDeviceCredentialIntent()`, you are requesting that the user authenticate by whatever means is appropriate. If a fingerprint is an option, the user will be able to use a fingerprint to authenticate.

This is cheap and easy, but it does not *require* that the user use a fingerprint. The user could type in their PIN or passphrase instead.

Using RxFingerprint

It is possible that you will want to use a fingerprint specifically. For example, perhaps you are using fingerprints as part of a multi-factor authentication system, where the user needs to provide some sort of passphrase for your app *and* authenticate with a fingerprint.

The simplest way to do this is to use one of the many third-party libraries that exist to simplify Android's fingerprint API. For example, the `DeviceAuth/FingerCheck` sample application uses RxFingerprint, which offers an RxJava-compatible API for authenticating a user via a registered fingerprint.

The first requirement for working with fingerprints explicitly is to request the `USE_FINGERPRINT` permission:

```
  <uses-permission android:name="android.permission.USE_FINGERPRINT" />
```

(from DeviceAuth/FingerCheck/app/src/main/AndroidManifest.xml)

This is not a `dangerous` permission, so you do not need to request it at runtime using `requestPermissions()`. But, you have to have it, so that users that investigate the full slate of permissions requested by your app will know that you wish to work with fingerprints.

RxFingerprint itself is just another artifact to request as part of your `dependencies`:

```
dependencies {
    implementation 'com.android.support:support-vector-drawable:27.0.2'
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    implementation 'io.reactivex.rxjava2:rxjava:2.1.7'
    implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'
    implementation 'com.mtramin:rxfingerprint:2.2.1'
}
```

(from DeviceAuth/FingerCheck/app/build.gradle)

When you want to authenticate the user, you can use `isAvailable()` and `authenticate()` `static` methods on the `RxFingerprint` class:

```
if (RxFingerprint.isAvailable(this)) {
  disposable=RxFingerprint.authenticate(this)
    .subscribe(this::onAuthResult,
      t -> {
        Log.e(getClass().getSimpleName(), "Exception authenticating", t);
      });
}
else {
  Toast.makeText(this, R.string.msg_not_available, Toast.LENGTH_LONG).show();
}
```

You need to call `isAvailable()` first. If this returns `false`, you cannot authenticate using a fingerprint, for any number of reasons, including:

- The device is running on less than API Level 23, which you can validate yourself using `Build.VERSION.SDK_INT`
- The device does not have a fingerprint sensor — you can also check this by calling `hasSystemFeature(PackageManager.FEATURE_FINGERPRINT)` on a `PackageManager` and seeing if it returns `true` or `false`
- The user has not registered a fingerprint

Assuming that there is a fingerprint eligible for authentication, `authenticate()` turns on the fingerprint sensor. There is no visible indication of this, and so you will need to do something in your app to let the user know that it is time for them to scan their fingerprint.

There are four possible outcomes. The unexpected one is some sort of crash. In that case, your RxJava stream will contain a `Throwable` representing the crash. In the code shown above, `subscribe()` routes that `Throwable` to some code that logs the message to LogCat (and, ideally, should do other things as well, to let the user know that something went wrong).

Everything else routes through a `FingerprintAuthenticationResult` object that is the output of the RxJava stream set up by `isAvailable()`. You can call `getResult()` on that to get an `enum` value that indicates what happened:

```
  private void onAuthResult(FingerprintAuthenticationResult authResult) {
    String msg=getString(R.string.msg_not_possible);

    switch (authResult.getResult()) {
      case FAILED:
        msg=getString(R.string.msg_failed);
        button.setImageDrawable(off);
        unsub();
```

```
      break;
    case HELP:
      msg=authResult.getMessage();
      break;
    case AUTHENTICATED:
      msg=getString(R.string.msg_authenticated);
      button.setImageDrawable(off);
      unsub();
      break;
  }

  Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
}
```

(from DeviceAuth/FingerCheck/app/src/main/java/com/commonsware/android/auth/finger/MainActivity.java)

If we get a `HELP` event, the user can try to scan their fingerprint again. The `getMessage()` method on the `FingerprintAuthenticationResult` provides a message to show the user to hint at what they might do to get a better scan. Otherwise, `AUTHENTICATED` means that the user successfully authenticated using a fingerprint, and `FAILED` means that the user failed authentication.

In all three cases, the fingerprint sensor is still active, and will remain so until we `dispose()` of the `Disposable` representing this RxJava subscription, handled here by the `unsub()` method:

```
private void unsub() {
  if (disposable!=null) {
    disposable.dispose();
    disposable=null;
  }
}
```

(from DeviceAuth/FingerCheck/app/src/main/java/com/commonsware/android/auth/finger/MainActivity.java)

In this sample app, the layout consists of a single `ImageButton`, showing a vector drawable of a stylized fingerprint. That fingerprint will show up in one of two colors: black normally, but blue when the fingerprint sensor is active. In `onCreate()`, we set up tinted editions of the vector drawable, plus hook up an `authenticate()` method to the button:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
```

```
    off=DrawableCompat.wrap(VectorDrawableCompat.create(getResources(),
      R.drawable.ic_fingerprint_black_24dp, null));
    off.setTint(getResources().getColor(android.R.color.black, null));

    on=DrawableCompat.wrap(VectorDrawableCompat.create(getResources(),
      R.drawable.ic_fingerprint_black_24dp, null));
    on.setTint(getResources().getColor(R.color.primary, null));

    button=findViewById(R.id.fingerprint);
    button.setImageDrawable(off);
    button.setOnClickListener(view -> authenticate());
  }
```

(from DeviceAuth/FingerCheck/app/src/main/java/com/commonsware/android/auth/finger/MainActivity.java)

`authenicate()` contains the `RxFingerprint` calls from earlier, but with some additional manipulation of the button drawable, to toggle it blue while the scan is going on and toggle it black if an exception occurs:

```
  private void authenticate() {
    unsub();

    if (RxFingerprint.isAvailable(this)) {
      button.setImageDrawable(on);
      disposable=RxFingerprint.authenticate(this)
        .subscribe(this::onAuthResult,
          t -> {
            Log.e(getClass().getSimpleName(), "Exception authenticating", t);
            button.setImageDrawable(off);
          });
    }
    else {
      Toast.makeText(this, R.string.msg_not_available, Toast.LENGTH_LONG).show();
    }
  }
```

(from DeviceAuth/FingerCheck/app/src/main/java/com/commonsware/android/auth/finger/MainActivity.java)

The layout sets the `ImageButton` to be approximately one inch square:

Figure 765: Finger Check Sample, As Initially Launched

When you tap that button, the fingerprint turns blue:

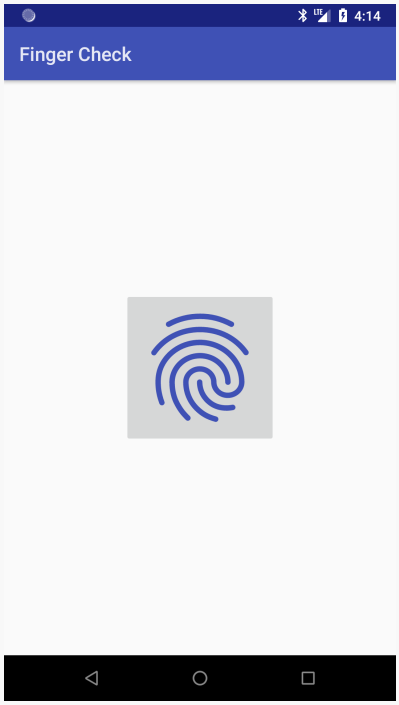


Figure 766: Finger Check Sample, As Initially Launched

From there, what happens depends on the user, but ideally involves a successful authentication:

Figure 767: Finger Check Sample, As Initially Launched

Directly

The `FingerprintManager` system service is the Android gateway to the fingerprint-specific APIs. It has only three methods, at least as of Android 8.1.

Two of these simply indicate whether or not you can bother using the third method:

- `isHardwareDetected()` returns `true` if the device has a fingerprint sensor that is set up for use by the Android SDK, `false` otherwise
- `hasEnrolledFingerprints()` returns `true` if the user has set up one or more fingerprints in Settings, `false` otherwise

If either of those methods return `false`, you cannot authenticate the user by fingerprint.

If they both return `true`, though, you can use the third method — `authenticate()` – to authenticate the user with a fingerprint.

`authenticate()` takes five parameters:

- A `CryptoObject`, which for simple authentication can be `null`
- A `CancellationSignal` object that you can use to cancel the attempt to authenticate
- Some currently-unused flags (pass in `0` for now)
- A callback object, to find out what is happening with the authentication attempt
- A `Handler`, if you want the callback methods to be called on some other thread (otherwise, pass `null` to be called back on the main application thread)

Your `AuthenticationCallback` implementation can be called with four methods:

- `onAuthenticationSucceeded()`, if the user did indeed pass the authentication
- `onAuthenticationFailed()`, if the sensor detected a fingerprint, but it was not one of the registered fingerprints (e.g., from some other person)
- `onAuthenticationHelp()`, when there was a problem in scanning the fingerprint, but the user can try again (e.g., the finger moved too much)
- `onAuthenticationError()`, when something goes wrong that does not represent a failure (i.e., mismatched fingerprint) and is not recoverable

RxFingerprint simply wraps all of this in an RxJava-friendly API.

Encrypting Data Using Fingerprints and Libraries

The `CryptoObject` that you can pass into `authenticate()` is tied to the device keystore for use with digitally signing and encrypting data securely. You do not have to use `FingerprintManager` to work with the keystore, but it is one option. However, as we will see in the chapter on the keystore, working with the keystore is a bit complex.

RxFingerprint

RxFingerprint offers simple `encrypt()` and `decrypt()` methods. Each return an `Observable` for you to be able to react to the results of the cryptography.

Both methods have three parameters in common:

1. An `EncryptionMethod` `enum` value, which is either `AES` (for AES-256 symmetric key encryption) or `RSA` (for public-key encryption)
2. The `Activity` requesting this work

3. A “key name”, which must be unique within your app, to identify the encryption key(s) that are generated and securely stored as part of this work

The advantage of `RSA` is that the user only needs to scan their fingerprint when data is decrypted. Encryption can occur at any time without a fingerprint. With `AES`, a fingerprint is needed for both encryption and decryption. However, `AES` should be a bit faster, particularly for larger amounts of data.

The `encrypt()` method takes a fourth parameter, which is a `String` representing the data to be encrypted. It is your job to marshal your data into a `String` by one means or another. What you get, reactively, from the `encrypt()` process is another `String`, representing the encrypted value. It is your job to save this somewhere, such as in a file.

The `decrypt()` method then works in reverse:

- Its fourth parameter is the encrypted `String`
- The result of the decryption is the original “plaintext” value that you supplied to `encrypt()` originally

the `DeviceAuth/FingerKey` sample application illustrates the use of these two methods.

The value that we are saving is a randomly-generated passphrase, for reasons that will become clearer later in this chapter. Specifically, it is 128 characters of random alphanumeric values, generated via a `SecureRandom` instance:

```
private char[] generatePassphrase() {
  char[] result=new char[128];

  for (int i=0; i<result.length; i++) {
    result[i]=BASE36_SYMBOLS.charAt(rng.nextInt(BASE36_SYMBOLS.length()));
  }

  return result;
}
```

(from DeviceAuth/FingerKey/app/src/main/java/com/commonsware/android/key/MainActivity.java)

Here, `BASE36_SYMBOLS` is simply a `String` of the candidate characters to use in the random passphrase:

```
private static final String BASE36_SYMBOLS="abcdefghijklmnopqrstuvwxyz0123456789";
```

(from DeviceAuth/FingerKey/app/src/main/java/com/commonsware/android/key/MainActivity.java)

`rng` is a `SecureRandom` instance:

```
  private final SecureRandom rng=new SecureRandom();
```

(from DeviceAuth/FingerKey/app/src/main/java/com/commonsware/android/key/MainActivity.java)

That passphrase is generated as part of starting up the activity and is held onto in a `passphrase` field.

The UI is the same as in the `FingerAuth` sample shown above, with a large fingerprint button. This time, when the user taps the button, we will encrypt the passphrase using RSA encryption, then turn around and decrypt the value. The decryption will require the user to scan their fingerprint.

The button is now tied to a `doTheWork` method, which confirms that we have fingerprint access, then calls `encrypt()` on the `RxFingerprint` instance:

```
private void doTheWork() {
  unsub();

  if (RxFingerprint.isAvailable(this)) {
    disposable=RxFingerprint.encrypt(EncryptionMethod.RSA, this, KEY_NAME,
      new String(passphrase))
      .subscribe(this::onEncResult,
        t -> {
          Log.e(getClass().getSimpleName(), "Exception authenticating", t);
          button.setImageDrawable(off);
        });
  }
  else {
    Toast.makeText(this, R.string.msg_not_available, Toast.LENGTH_LONG).show();
  }
}
```

(from DeviceAuth/FingerKey/app/src/main/java/com/commonsware/android/key/MainActivity.java)

Here, `KEY_NAME` is just a simple constant. The `passphrase` is held onto as a `char` array, so we need to convert that to a `String` before passing it to `encrypt()` as the value to be encrypted.

The `subscribe()` call on the `Observable` routes control to an `onEncResult()` method in normal cases, or logs an `Exception` to LogCat in case of a serious failure.

There are three possible outcomes from `onEncResult()`... in theory. In reality, it will depend on the encryption algorithm. `AES` requires fingerprint authentication for both encryption and decryption, and so there is a chance that the user fails the

fingerprint authentication step. `RSA` — which this sample app uses – does not require fingerprint authentication for encryption. This means that everything should succeed (a result of `AUTHENTICATED`, or we should get a `Throwable`.

So, if we are `AUTHENTICATED`, our data has been encrypted, and so we can get that encrypted value via `getEncrypted()` on the `FingerprintEncryptionResult`. In a real app, you would save this somewhere for long-term use. Here, we just turn right around and:

- Show the "on" state for the `ImageButton`, with a blue fingerprint
- Clean up the previous subscription via `unsub()`
- Pass the `getEncrypted()` value to `RxFingerprint.decrypt()` to decrypt this value

```
private void onEncResult(FingerprintEncryptionResult encResult) {
  if (encResult.getResult()==FingerprintResult.AUTHENTICATED) {
    button.setImageDrawable(on);
    unsub();

    String encryptedValue=encResult.getEncrypted();

    disposable=RxFingerprint.decrypt(EncryptionMethod.RSA, this, KEY_NAME,
      encryptedValue)
      .subscribe(this::onDecResult,
        t -> {
          Log.e(getClass().getSimpleName(), "Exception decrypting", t);
          button.setImageDrawable(off);
        });
  }
  else {
    Toast.makeText(this, "This was unexpected...", Toast.LENGTH_LONG).show();
  }
}
```

(from DeviceAuth/FingerKey/app/src/main/java/com/commonsware/android/key/MainActivity.java)

`decrypt()` takes the same first three parameters as does `encrypt()`:

1. The `EncryptionMethod` `enum` value corresponding with how you encrypted the value
2. The `Activity` requesting this work
3. The same "key name" that you used with `encrypt()`

The fourth parameter is the encrypted data to be decrypted.

`decrypt()` returns an `Observable` of `FingerprintDecryptionResult`. We use the same pattern as before, routing the normal case to an `onDecResult()` method and logging any exceptions to LogCat.

`onDecResult()` will need to deal with all three possible fingerprint scan outcomes (`FAILED`, `HELP`, and `AUTHENTICATED`), since any decryption requires a fingerprint scan. The first two of those scenarios we handle the same as in `FingerAuth`; the third one just confirms that our decrypted data matches the plaintext that we supplied to `encrypt()` back in the beginning:

```
private void onDecResult(FingerprintDecryptionResult decResult) {
  String msg=getString(R.string.msg_not_possible);

  switch (decResult.getResult()) {
    case FAILED:
      msg=getString(R.string.msg_failed);
      button.setImageDrawable(off);
      unsub();
      break;
    case HELP:
      msg=decResult.getMessage();
      break;
    case AUTHENTICATED:
      button.setImageDrawable(off);
      unsub();

      if (decResult.getDecrypted().equals(new String(passphrase))) {
        msg=getString(R.string.msg_match);
      }
      else {
        msg=getString(R.string.msg_mismatch);
      }

      break;
  }

  Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
}
```

(from DeviceAuth/FingerKey/app/src/main/java/com/commonsware/android/key/MainActivity.java)

In a real app, of course, you would use this decrypted data for something useful.

Whorlwind

Square — authors of a seemingly-infinite number of open source Android libraries — has their take on fingerprints and encryption, in the Whorlwind library.

This is simpler to use than RxFingerprint, if you do not need control over the encryption algorithm that is used or the storage location for the encrypted data. That is handled for you by Whorlwind. It uses the `RSA` algorithm, which avoids the need for fingerprint authentication for encrypting the data. And, it provides a `Storage` abstraction for persisting the encrypted data in a key-value store. Whorlwind ships with a `SharedPreferencesStorage`, or you can create another implementation of the `Storage` interface that meets your needs.

The biggest issue with Whorlwind is that version `1.0.1` — the current version as of January 2018 when this paragraph was written — uses RxJava 1.x. That makes it difficult to use in an app that otherwise uses RxJava 2.x. With luck, patches that upgrade the library to RxJava 2.x will be merged into a new release.

OK, So What Was *That* All About?

Encrypting a random string seems to be a bit esoteric. Developers are used to encrypting things of value, and a random string would not appear to be of value.

On its own, a random string is useless.

However, there are encrypted data stores for Android that rely on a passphrase, such as SQLCipher for Android. Those data stores have no means of integrating with a fingerprint scanner directly.

The encrypted random string, therefore, serves as a bridge between the fingerprint authentication — or, as we will see, the keystore — and the passphrase-based data store:

- We generate the long random string
- We use that string as the passphrase with the data store
- We encrypt that string using RxFingerprint, Whorlwind, etc.
- We save that encrypted string somewhere (inside a Whorlwind `Storage`, in a file, in `SharedPreferences`, etc.)

Later on, to access the passphrase-based data store, we:

- Load the encrypted string
- Decrypt it, which will involve user authentication
- Use the decrypted random string as our passphrase

The net result is an encrypted data store (e.g., SQLCipher for Android) with device-based authentication, instead of (or in addition to) a user-entered passphrase.

We will explore this pattern in greater detail in the next chapter.

Limitations of Fingerprint-Based Encryption

Under the covers, all of this winds up using the keystore for handling the encryption securely. Using the keystore has many benefits, but it also has some limitations, particularly when it comes to changes to the lockscreen. The next chapter has a section that reviews these limitations.

Keys and the Keystore

The Java Cryptography Architecture (JCA) has been available in Java since Jave 1.1, though it has expanded over the years. It gives us `KeyStore` objects that can manage cryptographic keys for a variety of symmetric and asymmetric ciphers. You can create keys, then use those keys to encrypt, decrypt, sign, and validate data. The actual algorithms that implement the cryptography come from service provider implementations (SPIs) that plug into the JCA. You do not need to know all of the details of the cryptography – you treat the algorithms mostly as a black box.

Android extends JCA by offering a `KeyStore` that is integrated into the Android architecture. In particular, on some Android 7.0+ devices, the keys can be tied to hardware, as part of the so-called Trusted Execution Environment (TEE, presumably named by a golfer). This dramatically reduces the likelihood of your keys somehow getting leaked to some malicious actor. The Android-supplied `KeyStore` also offers hooks to tie keys to device authentication, where you can require the user to authenticate the device before you can use the key to decrypt the data.

In this chapter, we will explore the basics of using this Android-specific `KeyStore`. The important word in the preceding sentence is "basics", as this is *not* a complete treatment of how to use the JCA. The JCA is part of standard Java; other educational resources can show you how to implement effective cryptography using the JCA.

Also note that many of the Android-specific APIs shown in this chapter have a `minSdkVersion` of 23.

Prerequisites

To understand this chapter, please read the preceding chapter on device authentication first. Also, the examples in this chapter make extensive use of RxJava.

Terminology

First, let's review some...ummmm... "key" terms that will be used in this chapter.

Keys

To encrypt, decrypt, sign, or validate some data, you need a key and a corresponding algorithm.

From a `javax.crypto` standpoint, the Java class that we use to represent keys is `SecretKey`. Whether this is the "real" key, or is merely an identifier to "key material" held elsewhere (e.g., in hardware), depends on the SPI and `KeyStore` implementation.

KeyStore

A `KeyStore`, as the name suggests, is a storage location for keys. There are several implementations of `KeyStore`, based on the storage location and format of the keys in the store.

In Android, we are particularly interested in one known as `AndroidKeyStore`. It offers two benefits:

1. The "key material" — the actual bytes representing the key — never make it into our application process. Instead, the cryptography is performed by a system process. The APIs that we use make it appear that everything is local, but in reality IPC is occurring under the covers to pass our data and key identifier to this system process. The benefit here is that even if our process gets attacked, the attacker cannot get the "key material" to be able to decrypt encrypted data away from this device.
2. On supported devices, and for supported key/algorithm combinations, the key material may "bound" to the device itself. This means that even the system process that normally does the cryptography does not have the bytes

representing the key itself. Instead, the cryptography is performed by dedicated hardware, as part of the TEE. The key material held in the `AndroidKeyStore` needs to be blended with secure data held in hardware to get the *real* key to use with the cryptography. As a result, even if an attacker is able to hack core system processes, the attacker will not be able to retrieve a key that can be used outside of this device.

KeyChain

If you rummage around the Android SDK, you will also see a separate `KeyChain` class. This offers a stripped down system similar to the `KeyStore`. However, the `KeyChain` is for device-wide keys, and as such is usually not what developers of an individual app need.

Getting a KeyStore

The `AndroidKeyStore` is merely the name of a JCA `KeyStore` that you obtain using standard `javax.crypto` APIs:

```
ks=KeyStore.getInstance("AndroidKeyStore");
ks.load(null);
```

The static `getInstance()` method returns an instance of the `KeyStore`, and `load()` is used to populate it. With traditional JCA `KeyStore` implementations, `load()` is used to load the keys from a file. The `AndroidKeyStore` does not need this, and so we can pass `null` into the `load()` method.

Note that these methods throw a variety of checked exceptions, and so you will need to be in position to catch those and do something useful. In practice, nothing should go wrong, as those checked exceptions are mostly for cases where:

- You provide an invalid `KeyStore` type name (e.g., `KeyStore.getInstance("this does not exist")`))
- There are problems loading the keys from a file (file not found, file corrupt, etc.)

Creating a Key

Android has a dedicated `KeyGenParameterSpec` class, with a corresponding `Builder`, that is used to create secret keys for symmetric encryption, stored in the `AndroidKeyStore`:

```
KeyGenParameterSpec spec=
  new KeyGenParameterSpec.Builder("thisIsMyKey",
    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
    .build();

KeyGenerator keygen=
  KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

keygen.init(spec);
SecretKey secretKey=keygen.generateKey();
```

Grafting the Android-specific `AndroidKeyStore` onto the JCA base results in a few oddities. For example, we do not need to do anything to save this key in the `AndroidKeyStore` — saving it is a side effect of creating the key using a `KeyGenerator` backed by the `AndroidKeyStore` (via the `getInstance()` `static` method).

Each of your app's keys has a name (`"thisIsMyKey"` in the above example). You will use this name to reference the key later on. This name needs to be unique for your app, though it does not need to be unique for the entire device.

Most of the methods on the `KeyGenParameterSpec.Builder` are tied to the JCA, such as the block modes, encryption paddings, and so forth. The above configuration – when coupled with `KEY_ALGORITHM_AES` when creating the `KeyGenerator` – is for `AES/CBC/PKCS7Padding`, which is a fairly typical symmetric key setup. Most of the details of setting up the `Builder`, therefore, are tied to the specific form of encrypting or digital signature that you wish to employ. The details of this are well out of scope for this book and are best left to educational resources dedicated to Java cryptography.

The JCA `KeyStore` is designed around symmetric keys (e.g., `SecretKey`) classes, though asymmetric encryption has been grafted onto the original `KeyStore` API, so RSA-style public-key encryption can also be performed using the `AndroidKeyStore`.

Tying the Key to Device Authentication

Part of the reason for an Android-specific `KeyGenParameterSpec` class is to be able to offer Android-specific features to extend the JCA. A prominent example of this is tying a generated key to device authentication, such that your app can only use the key if the user is currently authenticated.

The two primary `Builder` methods for this are:

- `setUserAuthenticationRequired(true)`, to indicate that the key being generated requires user authentication
- `setUserAuthenticationValidityDurationSeconds()`, to indicate how long we have, after user authentication, to be able to use the key, before the user needs to re-authenticate

The default validity period is, in effect, zero seconds. Every use of the key requires an immediately-preceding authentication. This may be excessive, and you will want to consider whether setting a longer validity period (e.g., one minute) is more appropriate for your situation.

So, in this sample, we generate a key with user authentication required, with a 60-second timeout:

```
KeyGenParameterSpec spec=
  new KeyGenParameterSpec.Builder("thisIsMyKey",
    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationValidityDurationSeconds(60)
    .build();

KeyGenerator keygen=
  KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

keygen.init(spec);
SecretKey secretKey=keygen.generateKey();
```

Later on, when we try to use the key for encryption, decryption, etc., we may get a `UserNotAuthenticatedException`. This means that the user has not authenticated against the device within the timeout period, and so we need to re-authenticate the user (e.g., `createConfirmDeviceCredentialIntent()`) before trying the

cryptographic operation again. The preceding chapter outlines how to use `createConfirmDeviceCredentialIntent()` to authenticate the user.

Learning More About Your Key

For keys created in the `AndroidKeyStore`, you can use a somewhat clunky set of APIs to find out more details about the key, in particular whether the key is backed by any sort of hardware security (e.g., the TEE). This information is available through an Android-specific `KeyInfo` class. Unfortunately, you have to get one of those by means of a `SecretKeyFactory` and a cast, as shown below:

```
SecretKey key=(SecretKey)ks.getKey(keyName, null);
KeyInfo info=
  (KeyInfo)SecretKeyFactory.getInstance(key.getAlgorithm(), "AndroidKeyStore")
    .getKeySpec(key, KeyInfo.class);

if (info.isInsideSecureHardware()) {
  Toast.makeText(this, "Key is inside secure hardware", Toast.LENGTH_LONG).show();
}
else {
  Toast.makeText(this, "Key is only secured by software", Toast.LENGTH_LONG).show();
}
```

Given the `SecretKey`, you can get the associated `SecretKeyFactory`, based on the algorithm and keystore name. That `SecretKeyFactory` has a `getKeySpec()` method, which takes the `SecretKey` and the spec class (`KeyInfo.class` in this case). That needs to be cast to a `KeyInfo`, this JCA code probably pre-dates Java's support for generics.

The sample code shown above checks one specific characteristic of the key: does the key reside inside of secure hardware. You get that from a call to `isInsideSecureHardware()`.

Encrypting Data

So, with all of that as background, let's look at actually encrypting some data using all of this stuff.

The `DeviceAuth/SecureNote` sample application has a single activity with a really big `EditText` widget, for you to type in a note. We will store that note in an encrypted form, using a key that requires device authentication.

And to do that, we will spend some time in a bodega.

Introducing RxKeyBodega

The `RxKeyBodega` class in this project implements a small RxJava-based API wrapped around the relevant `javax.crypto` and `android.security.keystore` classes. The idea is to isolate most of the "hard core" encryption logic in this class, but allow for composability, so that the calling app can control things like:

- Where the encrypted data comes from, and what happens to it after it is decrypted
- Where the encrypted data goes, after it is encrypted
- What thread is used for all of this

The only aspect that `RxKeyBodega` cannot handle itself is device authentication, since that involves a UI.

(this is far smaller than a key "store"; hence, a key "bodega")

Encrypting the Note

First, let's look at the encryption path. Our `MainActivity` not only has the really big `EditText`, but it has a "save" action bar item that, when clicked, will save the contents of the `EditText` in an encrypted form.

RxKeyBodega Client

That action bar item is tied to a `save()` method on `MainActivity`, which uses `RxKeyBodega` to set up an RxJava chain to process our encryption:

```
private void save() {
  final Context app=getApplicationContext();
  byte[] toEncrypt=note.getText().toString().getBytes(UTF8);

  RxKeyBodega.encrypt(toEncrypt, KEY_NAME, TIMEOUT_SECONDS)
    .subscribeOn(Schedulers.io())
    .map(result -> {
      File f=new File(app.getFilesDir(), FILENAME);

      RxKeyBodega.save(f, result);

      return f;
    })
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
```

```
      file -> Toast.makeText(MainActivity.this, R.string.saved, Toast.LENGTH_SHORT).show(),
      t -> {
        if (t instanceof UserNotAuthenticatedException) {
          requestAuth(REQUEST_SAVE);
        }
        else {
          Toast.makeText(MainActivity.this, t.getMessage(), Toast.LENGTH_LONG).show();
          Log.e(getString(R.string.app_name), "Exception saving encrypted file", t);
        }
      });
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

We will take a look at `RxKeyBodega.encrypt()` in the next section, but it returns an `Observable` of an `EncryptionResult` object, which itself is part of `RxKeyBodega`. We supply the data to be encrypted as a `byte` array, which we encode using `UTF8` from the `Editable` returned by the `EditText` and its `getText()` method. We pass that, along with a unique name for the encryption key to use and how long the device authentication timeout should be. That encryption key will be lazy-created if it does not already exist.

After routing this work to the `io()` thread, we use `map()` to process the `EncryptionResult`. We create a `File` pointing to where we want the encrypted data to be stored on internal storage, then call `RxKeyBodega.save()` to save the `EncryptionResult` to that file. This `File` is then supplied downstream to our subscriber lambda, which:

- Shows a `Toast` if everything seems OK
- Calls `requestAuth()` if we got a `UserNotAuthenticatedException`, indicating that we need the user to re-authenticate with the device before we can encrypt
- Shows an error `Toast` if something else went wrong

`requestAuth()` takes a request code as a parameter and kicks off device authentication using `createConfirmDeviceCredentialIntent()`:

```
private void requestAuth(int requestCode) {
  Intent i=
    mgr.createConfirmDeviceCredentialIntent("title", "description");

  if (i==null) {
    Toast.makeText(this, "No authentication required?!?",
      Toast.LENGTH_SHORT).show();
  }
  else {
    startActivityForResult(i, requestCode);
```

```
    }
  }
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

In `onActivityResult()`, if the user authenticated, we try `save()` again:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (resultCode==RESULT_OK) {
    if (requestCode==REQUEST_SAVE) {
      save();
    }
    else if (requestCode==REQUEST_LOAD) {
      load();
    }
  }
  else {
    Toast.makeText(this, R.string.sorry, Toast.LENGTH_SHORT).show();
    finish();
  }
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

If the user declined to authenticate, we show a `Toast` and `finish()` the activity, since there is nothing useful that we can do.

RxKeyBodega Implementation

Overall, `RxKeyBodega` follows the implementation approach used by `RxFingerprint`, profiled in the previous chapter. So, for example, the `encrypt()` method on `RxKeyBodega` does not encrypt anything itself, but rather creates an `Observable` — named `EncryptObservable` — that will handle the encryption work itself:

```
static Observable<EncryptionResult> encrypt(byte[] toEncrypt, String keyName,
                                            int timeout) {
  return Observable.create(new EncryptObservable(keyName, timeout, toEncrypt));
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

This way, the consumer of `RxKeyBodega` can control the thread on which the encryption work is performed.

`EncryptObservable` extends a `BodegaObservable` class. It lazy-initializes a `KeyStore`, where `KEYSTORE` is `"AndroidKeyStore"`, so we get access to the (potentially) hardware-backed keystore:

```
private abstract static class BodegaObservable {
  KeyStore ks;
  Exception initException;

  BodegaObservable() {
    try {
      ks=KeyStore.getInstance(KEYSTORE);
      ks.load(null);
    }
    catch (Exception e) {
      initException=e;
    }
  }
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

Nothing should go wrong in that initialization, but `getInstance()` and `load()` throw some checked exceptions. If one occurs, it is held onto in an `initException` field.

`EncryptObservable` implements the `ObservableOnSubscribe` interface, so the bulk of its logic goes into the `subscribe()` method, called when something eventually subscribes to this `Observable`:

```
  @Override
  public void subscribe(ObservableEmitter<EncryptionResult> emitter)
    throws Exception {
    if (initException==null) {
      createKey(keyName, timeout);

      SecretKey secretKey=(SecretKey)ks.getKey(keyName, null);
      Cipher cipher=Cipher.getInstance("AES/CBC/PKCS7Padding");
      SecureRandom rand=new SecureRandom();
      byte[] iv=new byte[BLOCK_SIZE];

      rand.nextBytes(iv);

      IvParameterSpec ivParams=new IvParameterSpec(iv);

      cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParams);
      emitter.onNext(new EncryptionResult(ivParams.getIV(), cipher.doFinal(toEncrypt)));
    }
    else {
      throw initException;
```

```
    }
  }
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

If `initException` is `null`, we throw it here, so that the exception works its way through the RxJava chain and can be picked up by a `Throwable` handler in `MainActivity`.

If `initException` is `null`, though, we start of by lazy-creating our key, in a `createKey()` method:

```
  private void createKey(String keyName, int timeout) throws Exception {
    KeyStore.Entry entry=ks.getEntry(keyName, null);

    if (entry==null) {
      KeyGenParameterSpec spec=
        new KeyGenParameterSpec.Builder(keyName,
          KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
          .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
          .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
          .setUserAuthenticationRequired(true)
          .setUserAuthenticationValidityDurationSeconds(timeout)
          .setRandomizedEncryptionRequired(false)
          .build();

      KeyGenerator keygen=
        KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTORE);

      keygen.init(spec);
      keygen.generateKey();
    }
  }
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

This uses the approach outlined earlier in this chapter, creating an `AES/CBC/PKCS7Padding` key. In particular, it is tied to device authentication, so the user will have had to entered their PIN, passcode, fingerprint, etc. within the timeout period, or else the encryption cannot be performed.

Once we are sure that we have a key, `subscribe()` creates a `Cipher` for that key, uses `doFinal()` to pass along the data to be encrypted, and passes that `doFinal()` response, along with an "initialization vector", to an `EncryptionResult` constructor.

The `EncryptionResult` is what gets emitted by this `Observable` and is what is picked up by `MainActivity` in the first step of its RxJava chain.

The `save()` utility method on `RxKeyBodega`, used by `MainActivity`, writes the initialization vector data and the encrypted data to the supplied file:

```
static void save(File f, EncryptionResult result)
  throws IOException {
  BufferedSink sink=Okio.buffer(Okio.sink(f));

  sink.write(result.iv);
  sink.write(result.encrypted);
  sink.close();
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

Some Quick Initialization Vector Notes

Some encryption schemes use an initialization vector (IV). This is a series of random bytes that serves as the initial input into the encryption algorithm. The bytes do not have to be secret, which is why it is safe for us to save them as part of the encrypted data, as we did in the `subscribe()` method of `EncryptObservable` above.

However, the bytes *do* have to be random. This has been a problem with Android.

Early versions of Android had a hard-coded default IV value, rather than randomly generating a value. This is bad, and it forced developers to have to create their own IV using `SecureRandom`.

Google apparently has fixed the default IV behavior. In fact, to use your own IV, you have to call `setRandomizedEncryptionRequired()` on the `KeyGenParameterSpec.Builder`, as part of creating your key in the `AndroidKeyStore`. If you fail to do this, when you go to supply your own IV bytes for encryption, you crash, with an exception whose message reads: “Caller-provided IV not permitted when encrypting”.

So, as of API Level 23, Google wants you to use their random IV implementation, rather than supply your own IV bytes. But, on older Android devices, you cannot rely on Google’s implementation.

So, now what?

The safest course of action is to do what we are doing in the `SecureNote` sample: provide a random IV and tell Android that we intend to do this, via the `setRandomizedEncryptionRequired()`. This way, even if some device manufacturer screws something up and their device winds up with non-random default IV values, you are covered.

Escaping If Device Is Insecure

Lazy-creating that key — in fact, this entire app — only makes sense if the device has a secure keyguard. Otherwise, we cannot have the key be tied to device authentication.

So, part of what we do in `onCreate()` is get a `KeyguardManager` and see if the keyguard is secure, using the techniques outlined in the preceding chapter:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  mgr=(KeyguardManager)getSystemService(KEYGUARD_SERVICE);

  if (mgr.isKeyguardSecure()) {
    note=findViewById(R.id.note);
    load();
  }
  else {
    Toast.makeText(this, R.string.insecure, Toast.LENGTH_LONG).show();
    finish();
  }
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

If it is not secure, we show a `Toast` and `finish()` our activity to leave the app.

Starting Decryption in onCreate()

If we do have a secure keyguard, we call a `load()` method, which will kick off loading the encrypted data and decrypting it.

You might wonder why we do not take a similar approach for encrypting and saving the note. Rather than force the user to click a “save” action bar item, we could call `save()` from `onDestroy()`, or perhaps from `onStop()`.

The problem is that we may need the user to re-authenticate, if they have not authenticated since before the timeout window. That requires us to start an activity and get a result. However, that is not safe to do once our activity is destroyed. So what happens is that the user presses BACK to exit the app, we determine that we need to authenticate the user, and then:

- We have to fail to save the note, possibly with an error message, or
- We can try to show the authentication activity, get a result, and then save the note, but that is quite likely to fail, given that our activity will be destroyed along the way

As a result, while we can automatically decrypt the note, we cannot automatically encrypt it.

Decrypting the Note

Now, let's turn our attention to the `load()` method.

RxKeyBodega Client

`load()` implements another RxJava `Observable` chain, to take what we wrote to the file, decrypt it, and show the results in the `EditText` for possible changes by the user:

```
private void load() {
  Observable.just(new File(getFilesDir(), FILENAME))
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .filter(File::exists)
    .map(RxKeyBodega::load)
    .flatMap(toDecrypt -> RxKeyBodega.decrypt(toDecrypt, KEY_NAME))
    .subscribe(bytes -> note.setText(new String(bytes, UTF8)),
      t -> {
        if (t instanceof UserNotAuthenticatedException) {
          requestAuth(REQUEST_LOAD);
        }
        else {
          Toast.makeText(MainActivity.this, t.getMessage(), Toast.LENGTH_LONG).show();
          Log.e(getString(R.string.app_name), "Exception loading encrypted file", t);
        }
      });
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

The `Observable` starts out with a `File` pointing to where the note is saved on internal storage. We then use the `filter()` operator to only continue if the file

exists — if it doesn't, that means we have no notes to load, and so the empty `EditText` is a fine starting point.

We then use a `static` `load()` method on `RxKeyBodega` to do the file I/O to read in the file. We pass the result from `load()` to the `RxKeyBodega.decrypt()` method, supplying the same `KEY_NAME` that we will want to use for encryption.

At that point, there are three possibilities:

1. Everything succeeds, in which case we get the decrypted bytes, so we can turn them into a `String` (using the `UTF8` charset) and apply that to the `EditText`
2. We get a `UserNotAuthenticatedException`, in which case we request that the user authenticate, and if that succeeds, `onActivityResult()` will try the `load()` again
3. We get some other exception, in which case we just show a `Toast` and log the exception

RxKeyBodega Implementation

The `load()` method on `RxKeyBodega` simply reads in the data that we wrote out:

```
static EncryptionResult load(File f) throws Exception {
  BufferedSource source=Okio.buffer(Okio.source(f));
  byte[] iv=source.readByteArray(BLOCK_SIZE);
  byte[] encrypted=source.readByteArray();

  source.close();

  return new EncryptionResult(iv, encrypted);
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

To read in the initialization vector, we need to know how many bytes it should be. That is based on the choice of encryption cipher, but the value is a constant for any given cipher. So, we get that in a `static` initialization block, and pray to the high heavens that JCA does not thrown an unexpected exception:

```
private static final int BLOCK_SIZE;

static {
  int blockSize=-1;
```

```
    try {
      blockSize=Cipher.getInstance("AES/CBC/PKCS7Padding").getBlockSize();
    }
    catch (Exception e) {
      Log.e("RxKeyBodega", "Could not get AES/CBC/PKCS7Padding cipher", e);
    }

    BLOCK_SIZE=blockSize;
  }
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

(a production-grade app would do something with the exception besides logging it to LogCat)

`load()` then wraps the initialization vector and encrypted data in an `EncryptionResult`, which becomes input for the `decrypt()` method, which turns around and hands it to a `DecryptObserverable`:

```
static Observable<byte[]> decrypt(EncryptionResult toDecrypt, String keyName) {
  return Observable.create(new DecryptObservable(keyName, toDecrypt));
}
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

`DecryptObservable` then uses JCA to decrypt the encrypted data, using the chosen `Cipher` and initialization vector, emitting the decrypted data:

```
private static class DecryptObservable extends BodegaObservable implements
  ObservableOnSubscribe<byte[]> {
  final private String keyName;
  final private EncryptionResult toDecrypt;

  private DecryptObservable(String keyName, EncryptionResult toDecrypt) {
    this.keyName=keyName;
    this.toDecrypt=toDecrypt;
  }

  @Override
  public void subscribe(ObservableEmitter<byte[]> emitter)
    throws Exception {
    if (initException==null) {
      SecretKey secretKey=(SecretKey)ks.getKey(keyName, null);
      Cipher cipher=Cipher.getInstance("AES/CBC/PKCS7Padding");

      cipher.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(toDecrypt.iv));
      emitter.onNext(cipher.doFinal(toDecrypt.encrypted));
    }
    else {
      throw initException;
    }
```

```
    }
  }
```

(from DeviceAuth/SecureNote/app/src/main/java/com/commonsware/android/auth/note/RxKeyBodega.java)

Time-Limited Device Authentication

In the previous chapter, we saw the use of `createConfirmDeviceCredentialIntent()` to force the user to re-authenticate before proceeding with something in your app. As noted then, the problem is that `createConfirmDeviceCredentialIntent()` always forces re-authentication, even if the user authenticated just moments ago (e.g., just unlocked their device).

However, for working with `AndroidKeyStore`-backed keys, we can require device authentication... but with a timeout. That way, the user only needs to re-authenticate if they have not authenticated recently, for whatever value of "recently" we want when setting up the key.

So, if you want to use `createConfirmDeviceCredentialIntent()`, but you want a timeout, one solution is silly, but works: encrypt something with a timeout-enabled, authenticated key. What you encrypt does not matter — you can throw away the results of the encryption. You are simply leveraging the timeout facility to let you know whether authentication is really needed or not.

The `DeviceAuth/SecureTimeout` sample application demonstrates this. It is based on the `SecureCheck` sample from the preceding chapter, but now uses the encryption technique outlined above.

In `onCreate()`, we set up the `AndroidKeyStore`:

```
    try {
      ks=KeyStore.getInstance(KEYSTORE);
      ks.load(null);
    }
    catch (Exception e) {
      Toast.makeText(this, "Ummm... this shouldn't happen", Toast.LENGTH_LONG).show();
      Log.e(getClass().getSimpleName(), "Exception initializing keystore", e);
    }
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

We only enable the authentication action bar item if we have a secure keyguard, as otherwise we cannot set up the key that we want to use:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);
  menu.findItem(R.id.auth).setEnabled(mgr.isKeyguardSecure());

  return super.onCreateOptionsMenu(menu);
}
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

As before, tapping that action bar item invokes an `authenticate()` method. Before, that just blindly called `createConfirmDeviceCredentialIntent()` and forced authentication. Now... it's a bit different:

```
private void authenticate() {
  try {
    createKeyForTimeout();
  }
  catch (Exception e) {
    Toast.makeText(this, "Could not create the key", Toast.LENGTH_LONG).show();
    Log.e(getClass().getSimpleName(), "Exception creating key", e);
    return;
  }

  if (needsAuth(false)) {
    Intent i=
      mgr.createConfirmDeviceCredentialIntent("title", "description");

    if (i==null) {
      Toast.makeText(this, "No authentication required!",
        Toast.LENGTH_SHORT).show();
    }
    else {
      startActivityForResult(i, REQUEST_CODE);
    }
  }
}
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

We start by invoking the same sort of lazy-create-the-key logic that we used in `SecureNote`, this time in a `createKeyForTimeout()` method:

```
private void createKeyForTimeout() throws Exception {
  KeyStore.Entry entry=ks.getEntry(KEY_NAME, null);

  if (entry==null) {
    KeyGenParameterSpec spec=
      new KeyGenParameterSpec.Builder(KEY_NAME,
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
```

```
        .setUserAuthenticationRequired(true)
        .setUserAuthenticationValidityDurationSeconds(TIMEOUT_SECONDS)
        .build();

    KeyGenerator keygen=
      KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTORE);

    keygen.init(spec);
    keygen.generateKey();
  }
}
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

If that works, we call a `needsAuth()` method to see if we need authentication or not:

```
private boolean needsAuth(boolean isRecheck) {
  boolean result=false;

  try {
    SecretKey secretKey=(SecretKey)ks.getKey(KEY_NAME, null);
    Cipher cipher=Cipher.getInstance("AES/CBC/PKCS7Padding");

    cipher.init(Cipher.ENCRYPT_MODE, secretKey);
    cipher.doFinal(POINTLESS_DATA);

    if (!isRecheck) {
      Toast.makeText(this, "Already authenticated!", Toast.LENGTH_LONG).show();
    }
  }
  catch (UserNotAuthenticatedException e) {
    result=true;
  }
  catch (KeyPermanentlyInvalidatedException e) {
    Toast.makeText(this, "You reset the lock screen!",
      Toast.LENGTH_LONG).show();
  }
  catch (Exception e) {
    Toast.makeText(this, "Could not validate the key", Toast.LENGTH_LONG).show();
    Log.e(getClass().getSimpleName(), "Exception validating key", e);
  }

  return result;
}
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

This just encrypts a pointless bit of data:

```
private static final byte[] POINTLESS_DATA=new byte[] {1, 2, 3};
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

There are several possible outcomes of this. The biggest one is that we could catch a `UserNotAuthenticatedException`, meaning that the user has not authenticated within our timeout. In that case, `needsAuth()` returns `true`, and `authenticate()` uses `createConfirmDeviceCredentialIntent()` to trigger authentication. `onActivityResult()` then calls `needsAuth()` again, to confirm that we did successfully authenticate the user:

```
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==REQUEST_CODE) {
    if (resultCode==RESULT_OK) {
      if (needsAuth(true)) {
        Toast.makeText(this, "Good authentication... but still needs auth?",
          Toast.LENGTH_SHORT).show();
      }
      else {
        Toast.makeText(this, "Authenticated!", Toast.LENGTH_SHORT).show();
      }
    }
    else {
      Toast.makeText(this, "WE ARE UNDER ATTACK!", Toast.LENGTH_SHORT).show();
    }
  }
}
```

(from DeviceAuth/SecureTimeout/app/src/main/java/com/commonsware/android/auth/check/MainActivity.java)

The `boolean` parameter to `needsAuth()` simply indicates whether we are checking on the first or second pass. If we can successfully encrypt the pointless data, we know that the user has authenticated within our timeout period. If it's the first pass through `needsAuth()`, we show a `Toast` here, otherwise we show it in `onActivityResult()`.

This sample specifically checks for `KeyPermanentlyInvalidatedException`. This will be thrown if we *had* a key from before, but the user reset their lock screen, and that key is now no longer valid. If there was data encrypted using that key... the user is now in big trouble. From a JCA standpoint, you should be able to use `deleteEntry()` on the `KeyStore` to remove the old key under your key name and create a new one, if desired.

Encrypting Passphrases

The `AndroidKeyStore` is good for some scenarios, such as your own encrypted files. However, other things will need passphrases and do not integrate with the JCA. A prominent example is SQLCipher for Android. SQLCipher is an extension to SQLite, and SQLite is not tied to Android. Hence, SQLCipher is not tied to

Android, and the SQLCipher for Android distribution is focused more on providing a near-clone of the Android SQLite classes (e.g., `SQLiteDatabase`).

However, you can integrate anything that takes a passphrase with the `AndroidKeyStore` – and, by extension, with fingerprint-based device authentication — by generating the passphrase and encrypting it. This works akin to encrypting notes, as seen in the earlier example, in that you are encrypting and decrypting from files. However, rather than the files being the actual data, they are merely the keys with which to access that actual data, which has its own security solution.

The `DeviceAuth/CipherNote` sample application is a clone of the `SecureNote` sample. There, the note was encrypted and saved as a file. In this sample, the note is saved in SQLCipher for Android, with a generated passphrase being encrypted and saved as a file. In this case, this approach is overkill. However, there are plenty of scenarios where you need a SQLite-like solution for querying and such, but you also want to give the user the option of tying their security to device authentication, instead of having to type in a passphrase themselves.

From a UI standpoint, `CipherNote` works the same as `SecureNote`:

- The existing content is decrypted when the activity starts up, if that content exists, and possibly involving device authentication if needed
- When the user taps the “save” action bar item, the content is saved in an encrypted form

A NoteRepository

We start off with a simple `SQLiteOpenHelper` subclass named `DatabaseHelper`:

```
package com.commonsware.android.auth.note;

import android.content.Context;
import net.sqlcipher.database.SQLiteDatabase;
import net.sqlcipher.database.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="note.db";
  private static final int SCHEMA=1;

  DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, SCHEMA);

    SQLiteDatabase.loadLibs(context);
  }

  @Override
```

```
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE note (_id INTEGER PRIMARY KEY AUTOINCREMENT, content TEXT);");
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    throw new RuntimeException("How did we get here?");
  }
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/DatabaseHelper.java)

This sets up a trivial table (`note`) with a primary key and the content. It also initializes SQLCipher for Android, by means of `SQLiteDatabase.loadLibs(context)`.

That is then wrapped in a `NoteRepository`, which provides us with an RxJava-based API for loading and saving notes from the encrypted database, given a passphrase. As with many repository-style classes — and as with many things that work with SQLite or SQLCipher for Android — `NoteRepository` is a singleton. Its sole instance is lazy-created, as it needs a `Context` for use with `DatabaseHelper`:

```
  private static volatile NoteRepository INSTANCE;
  private SQLiteDatabase db;

  private synchronized static NoteRepository init(Context ctxt, char[] passphrase) {
    if (INSTANCE==null) {
      INSTANCE=new NoteRepository(ctxt.getApplicationContext(), passphrase);
    }

    return INSTANCE;
  }

  private synchronized static NoteRepository get() {
    return INSTANCE;
  }

  private NoteRepository(Context ctxt, char[] passphrase) {
    DatabaseHelper helper=new DatabaseHelper(ctxt);

    db=helper.getWritableDatabase(passphrase);
  }
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/NoteRepository.java)

The `NoteRepository` holds onto the `SQLiteDatabase` opened from the `DatabaseHelper`. More importantly, it does *not* hold onto the passphrase. Ideally, that passphrase should be cleared out of memory after the database is opened, as SQLCipher for Android no longer needs it, and the longer the passphrase is in memory, the more likely it is that somebody is going to find a way to extract it.

Our model object representing the note is named `Note`:

```
static class Note {
  final long id;
  final String content;

  private Note(long id, String content) {
    this.id=id;
    this.content=content;
  }
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/NoteRepository.java)

Loading the `Note` is reactive. We have a `static` `load()` method that returns an `Observable` based on a `LoadObservable`:

```
static Observable<Note> load(Context ctxt, char[] passphrase) {
  return Observable.create(new LoadObservable(ctxt, passphrase));
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/NoteRepository.java)

`LoadObservable` does as its name suggests: it loads the note from the database:

```
private static class LoadObservable implements ObservableOnSubscribe<Note> {
  private final Context app;
  private final char[] passphrase;

  LoadObservable(Context ctxt, char[] passphrase) {
    this.app=ctxt.getApplicationContext();
    this.passphrase=passphrase;
  }

  @Override
  public void subscribe(ObservableEmitter<Note> e) throws Exception {
    Cursor c=NoteRepository.init(app, passphrase).db
      .rawQuery("SELECT _id, content FROM note", null);

    if (c.isAfterLast()) {
      e.onNext(EMPTY);
    }
    else {
      c.moveToFirst();
      e.onNext(new Note(c.getLong(0), c.getString(1)));
      Arrays.fill(passphrase, '\u0000');
    }

    c.close();
```

```
    }
  }
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/NoteRepository.java)

While `LoadObservable` holds onto the passphrase, it does not make its own copy. Instead, it uses `Arrays.fill()` to clear out that `char` array as part of loading the note.

If there are no rows in the database, that means this was the first run of the app (or the user cleared the app's data). RxJava does not like an `ObservableEmitter` emitting a `null` value, so we have a magic `EMPTY` constant `Note` to use that signifies that we had no note:

```
  private static final Note EMPTY=new Note(-1, null);
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/NoteRepository.java)

Otherwise, this code is unremarkable: it queries the database, gets the values out of the `Cursor` to create a `Note`, and emits the `Note`.

There is a corresponding `save()` method. It is not reactive, but instead is designed to be added to some external RxJava chain. It takes the existing `Note` and the revised content from the `EditText`, either inserts or updates the database with the content, and returns a fresh `Note` instance:

```
  static Note save(Note note, String content) {
    ContentValues cv=new ContentValues(1);

    cv.put("content", content);

    if (note==EMPTY) {
      long id=NoteRepository.get().db.insert("note", null, cv);

      return new Note(id, content);
    }
    else {
      NoteRepository.get().db.update("note", cv, "_id=?",
        new String[]{String.valueOf(note.id)});

      return new Note(note.id, content);
    }
  }
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/NoteRepository.java)

This way, clients can blindly request a `save()`, and the repository can determine if that requires an insert or an update, based on whether we are starting with the `EMPTY` note or not.

RxPassphrase

This sample modifies `RxKeyBodega` from earlier into `RxPassphrase`. Here, we have a reactive API to get our passphrase for use with our SQLCipher for Android database... including lazy-creating (and encrypting) that passphrase if it does not already exist.

The public API is a simple static `get()` method. It takes the file to use for storing the passphrase, the key name for our key in the `AndroidKeyStore`, and our desired authentication timeout as parameters. It just creates a `PassphraseObservable` to do the real work:

```
static Observable<char[]> get(File encryptedFile, String keyName, int timeout) {
  return Observable.create(new RxPassphrase.PassphraseObservable(encryptedFile, keyName, timeout));
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/RxPassphrase.java)

`subscribe()` initializes our `KeyStore`, then sees if the encrypted passphrase file exists, branching to `load()` or `create()` methods accordingly:

```
  @Override
  public void subscribe(ObservableEmitter<char[]> emitter) throws Exception {
    KeyStore ks=KeyStore.getInstance(KEYSTORE);

    ks.load(null);

    if (encryptedFile.exists()) {
      load(ks, emitter);
    }
    else {
      create(ks, emitter);
    }
  }
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/RxPassphrase.java)

`create()` first creates a 128-character passphrase, using the same base36 algorithm used in an example from the preceding chapter:

```
  private void create(KeyStore ks, ObservableEmitter<char[]> emitter)
    throws Exception {
```

```
    SecureRandom rand=new SecureRandom();
    char[] passphrase=new char[128];

    for (int i=0; i<passphrase.length; i++) {
      passphrase[i]=BASE36_SYMBOLS.charAt(rand.nextInt(BASE36_SYMBOLS.length()));
    }

    createKey(ks, keyName, timeout);

    SecretKey secretKey=(SecretKey)ks.getKey(keyName, null);
    Cipher cipher=Cipher.getInstance("AES/CBC/PKCS7Padding");
    byte[] iv=new byte[BLOCK_SIZE];

    rand.nextBytes(iv);

    IvParameterSpec ivParams=new IvParameterSpec(iv);

    cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParams);

    byte[] toEncrypt=toBytes(passphrase);
    byte[] encrypted=cipher.doFinal(toEncrypt);

    BufferedSink sink=Okio.buffer(Okio.sink(encryptedFile));

    sink.write(iv);
    sink.write(encrypted);
    sink.close();

    emitter.onNext(passphrase);
  }
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/RxPassphrase.java)

`create()` then lazy-creates our key in our `KeyStore`, using the same `createKey()` method as was used previously:

```
  private void createKey(KeyStore ks, String keyName, int timeout)
    throws Exception {
    KeyStore.Entry entry=ks.getEntry(keyName, null);

    if (entry==null) {
      KeyGenParameterSpec spec=
        new KeyGenParameterSpec.Builder(keyName,
          KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
          .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
          .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
          .setUserAuthenticationRequired(true)
          .setUserAuthenticationValidityDurationSeconds(timeout)
          .setRandomizedEncryptionRequired(false)
          .build();

      KeyGenerator keygen=
        KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTORE);
```

```
      keygen.init(spec);
      keygen.generateKey();
    }
  }
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/RxPassphrase.java)

We then create a random initialization vector, and use that to help create our `Cipher` for encryption. We encrypt the passphrase and IV bytes, writing the results to the designated file, before returning the cleartext passphrase. That passphrase can then be passed to the `NoteRepository` for the purposes of creating our encrypted database.

Note that encryption might throw a `UserNotAuthenticatedException`, which the client needs to catch and route through the `createConfirmDeviceCredentialIntent()`-based UI for authenticating the user.

If our encrypted passphrase file already exists, we can just open and decrypt it:

```
  private void load(KeyStore ks, ObservableEmitter<char[]> emitter)
    throws Exception {
    BufferedSource source=Okio.buffer(Okio.source(encryptedFile));
    byte[] iv=source.readByteArray(BLOCK_SIZE);
    byte[] encrypted=source.readByteArray();

    source.close();

    SecretKey secretKey=(SecretKey)ks.getKey(keyName, null);
    Cipher cipher=Cipher.getInstance("AES/CBC/PKCS7Padding");

    cipher.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(iv));

    byte[] decrypted=cipher.doFinal(encrypted);
    char[] passphrase=toChars(decrypted);

    emitter.onNext(passphrase);
  }
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/RxPassphrase.java)

We do not use `createKey()` here, as by definition our key should already exist, as we used it to encrypt the file. If for some reason our *key* is lost, then our data is lost, and we have to start over from scratch anyway.

Using the NoteRepository and RxPassphrase

The overall flow of `MainActivity` has not changed: we still call `load()` from `onCreate()` and still call `save()` from the action bar item click. Merely their implementations have changed, to blend `NoteRepository` and `RxPassphrase`.

`load()` starts an RxJava chain by asking `RxPassphrase` to `get()` the passphrase:

```
private void load() {
  final Context app=getApplicationContext();
  File encryptedFile=new File(getFilesDir(), FILENAME);

  RxPassphrase.get(encryptedFile, KEY_NAME, TIMEOUT_SECONDS)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .flatMap(chars -> NoteRepository.load(app, chars))
    .subscribe(this::onNoteReady,
      t -> {
        if (t instanceof UserNotAuthenticatedException) {
          requestAuth(REQUEST_LOAD);
        }
        else {
          Toast.makeText(MainActivity.this, t.getMessage(), Toast.LENGTH_LONG).show();
          Log.e(getString(R.string.app_name), "Exception loading encrypted file", t);
        }
    });
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

That passphrase is then used with `NoteRepository.load()` to get the note from the encrypted database, using `flatMap()` to attach the `Observable` from `NoteRepository.load()` onto the existing chain. Then, if everything succeeds, we call `onNoteReady()` to hold onto the `Note` object and populate the `EditText`:

```
private void onNoteReady(NoteRepository.Note note) {
  this.note=note;
  textarea.setText(note.content);
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

If we get a `UserNotAuthenticatedException`, we go through the same `requestAuth()` as before, triggering authentication. In `onActivityResult()`, we call `load()` again to try to get the `Note`, now that the user has authenticated.

The passphrase is stored on internal storage, in `getFilesDir()` (`new File(getFilesDir(), FILENAME)`). Hence, in principle, it will only get deleted if the database itself gets deleted, such as the user choosing “Clear Data” for our app

in Settings. This is important, as our data will be lost if *either* the database or the encrypted passphrase file are lost.

`save()` does not need to use `RxPassphrase`, as `NoteRepository` should already have the open database. So, we can just use `NoteRepository.save()` to persist the note:

```
private void save() {
  Observable.just(textarea.getText().toString())
    .map(content -> NoteRepository.save(note, content))
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::onNoteSaved,
      t -> {
        if (t instanceof UserNotAuthenticatedException) {
          requestAuth(REQUEST_LOAD);
        }
        else {
          Toast.makeText(MainActivity.this, t.getMessage(), Toast.LENGTH_LONG).show();
          Log.e(getString(R.string.app_name), "Exception loading encrypted file", t);
        }
      });
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

Here, `onNoteSaved()` just shows the "Saved!" `Toast`, plus calls `onNoteReady()` to ensure that we have the `Note` for any subsequent `save()` call:

```
private void onNoteSaved(NoteRepository.Note note) {
  Toast.makeText(this, R.string.saved, Toast.LENGTH_LONG).show();
  onNoteReady(note);
}
```

(from DeviceAuth/CipherNote/app/src/main/java/com/commonsware/android/auth/note/MainActivity.java)

A Key(Store) Limitation

The keys in the `AndroidKeyStore` are tied to a secure keyguard. That is why these samples check for a secure keyguard before proceeding.

But what if the user *had* a secure keyguard, then downgrades to merely swipe-to-unlock?

When they downgrade, they should get a system warning that they will lose some information. In particular, on fingerprint-enabled devices, they should get a note about losing stored fingerprints. However, the system warning may not be a sufficient deterrent, and the user may downgrade their keyguard security anyway.

If that happens, your keys in the `AndroidKeyStore` get wiped out. Even if the user turns right around and sets up a secure keyguard again, whatever had been in the `AndroidKeyStore` is gone, and anything encrypted with one of those keys will be unrecoverable.

Changing between different types of security — such as switching between a PIN and a passphrase — is fine and will not affect the `AndroidKeyStore`.

Miscellaneous Security Techniques

This chapter outlines some additional security measures that you can consider for your applications that do not necessarily warrant a full chapter on their own at this time.

In other words, it's just a pile of interesting security stuff.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. In addition, you should review the app signing chapter if you are unfamiliar with the signing process.

Public Key Validation

We sign our apps with signing keys all the time. By default, we are signing with a so-called "debug signing key", created automatically by the build tools. For production, we sign with a different signing key. The primary use of that signing key is to determine equivalence of authorship:

- Is this APK, representing an upgrade to an already-installed app, signed by the same signing key that signed that app?
- Is this APK, that requests firmware-defined `signature`-level permissions, signed by the same signing key that signed the firmware?

However, as it turns out, information about the public key that signed an APK is visible to us, for our own APK as well as for any other APK on the device. We can leverage that to help determine whether a given APK was signed by something we

recognize. This goes above and beyond using Android's built-in signature-based defenses (e.g., using a custom `signature`-level permission).

Scenarios

There are several scenarios in which we might imagine that we could employ our own public key validation. How well the technique will work, though, depends on what we are checking and the nature of the attack we are defending against.

Checking Yourself

You might consider checking your own app's public key. After all, if your app is not signed with your production signing key, something very strange is going on, and the natural reaction is that "something strange" is unlikely to be a good thing for you.

However, there are some issues here.

First and foremost, checking your own signing key assumes that whatever caused you to *not* be signed by that key did not also modify your validation algorithm. For example, suppose that you validate your signing key to determine if somebody perhaps reverse-engineered and modified your app, perhaps to remove some license checks. This will only catch an attacker that removed the licensing checks and did not *also* remove your signature validation, or modify the validation to use the attacker's signing key. While it is possible that an attacker will modify one part but not another, it remains unclear how well this defense will work in practice.

Also, bear in mind that you, as a developer, may be opting into services that intentionally change your app's signature. Various providers will "wrap" your app, whether for interstitial ad banners or for quasi-DRM. There are three possible ways that they wrap your app:

1. They sign it with their signing key, which means that your runtime validation of the key will fail, as your app is now signed by their key, not yours. This is also *very* risky, as if for whatever reason you are no longer able to use their service (e.g., they go out of business), you may have difficulty in upgrading your app, as you will not have the right key to use.
2. They sign it with *your* signing key, either one that you upload, or one that they generate for you. In this case, your runtime public key validation logic could still work. On the other hand, now this other firm is perfectly capable

of upgrading your app, or shipping other apps, signed with your production signing key, and this has its own set of risks.

3. They allow you to download the "wrapped" app and have you sign it yourself with your own signing key. This is the best alternative from a security standpoint, but it is the most tedious, as now you have additional work to do to publish your app.

Checking Arbitrary Other Apps

What will tend to be more reliable is to check *other* applications' public keys. While they might have been cracked, it is unlikely that the same attacker also attacked *your* app, and so you can help detect problems in others.

For example, let us consider a specific scenario: a client-side JAR for integration to a third-party app.

This book outlines many forms of IPC, from content providers to remote services to broadcast `Intent` objects. If you are creating an app that offers such IPC endpoints, you may wish to consider also shipping a JAR to make using those endpoints a bit easier. You might create a library that handles all of the details of sending commands to your remote service, or you might create a library that provides a wrapper around the AIDL-generated Java proxy classes for remote binding.

Another thing such a JAR could do is check the integrity of your app. The JAR's code is in the client's app, not yours, and while your app might be cracked, the client's app might not. You could check the validity of the public key of your own app from the client's app, and fail if there is a detected problem.

This might be especially important depending upon the nature of the app and the JAR that is providing access to it. If the app is an app offering on-device payments (e.g., a Google Wallet sort of app), and the app offers an API for other apps to do payments, it is fairly important that those other apps can trust the payment app. By checking the public key, your JAR can help provide that level of trust… or at least ensure that nobody else has done something specifically to degrade that trust.

This is particularly important for avoiding device-hosted man-in-the-middle attacks on your IPC from client apps to your app. In an ideal world, you would only allow IPC via `signature`-level permissions, but that will not work in cases where third parties are writing the clients.

If your IPC is based upon a service (command pattern or binding pattern), if multiple service implementations all advertise the same `<intent-filter>`, Android needs to decide which service will handle the request. First, it will take into account the `android:priority` value on the `<intent-filter>` (even though this behavior is currently undocumented). For multiple services with the same priority (e.g., no priority specified), the first one that was installed will be the one that is chosen. In either case, the client has no way to know, short of examining the service's public key, whether the service that will respond to the requests for IPC is the legitimate service or something else advertising that it supports the same `Intent` action. Even with Android 5.0 blocking your ability to bind via an implicit `Intent`, you wind up with the same sorts of problems when you use `resolveService()` to try to determine the `ComponentName` of the service to make an explicit `Intent` for it.

The Easy Solution: SignatureUtils

The author of this book has published the CWAC-Security library. Among other things, this library has a `SignatureUtils` class that makes it relatively easy for you to compare the signature of some Android app to a known good value.

All you need to do is call the static `getSignatureHash()` method, supplying some `Context` (any will do) and the package name of the app that you wish to check. This will return the SHA-256 hash of the signing key of the app, as a set of capitalized, colon-delimited hex values.

You can get the same sort of hash by running the Java 7 version of `keytool`. Hence, if the app you wish to test is another one of yours, perhaps signed with a different signing key, you can use `keytool` to get the value to compare with the result of `getSignatureHash()`. Or, during development, create a little utility app that will dump the `getSignatureHash()` value for the third-party app, and run it on a device containing a known good version of that app (i.e., one that does not appear to have been replaced by malware).

Ideally, over time, we will be able to get app developers to publish their SHA-256 hashes on their Web sites, as another means of getting a known value of the hash to compare at runtime.

If you determine that `getSignatureHash()` does *not* return the right value, this means that the app that is installed on the device is written by somebody other than the app's original author. Often times, this will mean the app has malware in it. It is up to you to determine how you wish to respond to this scenario:

- Alert the user?
- Send data back to your server, or to your analytics collection point, with details of the bad APK?
- Block usage of your app, or usage of features that depend upon the flawed third party?
- Something else?

Examining Public Keys

Under the covers, `SignatureUtils` uses `PackageManager` and related classes to examine what they somewhat erroneously refer to as "signatures". The `MiscSecurity/SigDump` sample project will allow us to browse the list of installed packages, see a decoded public key on the screen for a package that we select, plus dump the "signature" as a binary file for later comparison using another app.

The UI Structure

In this sample, we use a `SlidingPaneLayout` for a master-detail pattern presentation, as was demonstrated in the chapter on dealing with multiple screen sizes. The "master" fragment will be the list of packages; the "detail" fragment will be the decoded public key for the selected package.

The master fragment is implemented as `PackagesFragment`. It implements a typical `ListFragment` for use with the master/detail pattern, utilizing the `activated` state to show the context for the detail fragment. The detail will be `SignatureFragment`, which will display portions of the decoded public key in a `TableLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:shrinkColumns="1"
  android:stretchColumns="1">

  <TableRow>

    <TextView
      android:layout_gravity="center"
      android:layout_margin="4dp"
      android:text="@string/subject"
      android:textStyle="bold"/>

    <TextView android:id="@+id/subject"/>
```

```
  </TableRow>

  <TableRow>

    <TextView
      android:layout_gravity="center"
      android:layout_margin="4dp"
      android:text="@string/issuer"
      android:textStyle="bold"/>

    <TextView android:id="@+id/issuer"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_gravity="center"
      android:layout_margin="4dp"
      android:text="@string/valid_between"
      android:textStyle="bold"/>

    <TextView android:id="@+id/valid"/>
  </TableRow>

</TableLayout>
```

(from MiscSecurity/SigDump/app/src/main/res/layout/sig.xml)

Listing the Packages

Our `PackagesFragment` needs the list of packages to display. It expects the hosting activity to supply that, by using the contract pattern, and having a `getPackageList()` method on its `Contract`:

```
  interface Contract {
    void onPackageSelected(PackageInfo pkgInfo);

    List<PackageInfo> getPackageList();
  }
```

(from MiscSecurity/SigDump/app/src/main/java/com/commonsware/android/signature/dump/PackagesFragment.java)

The hosting activity — `MainActivity` — retrieves a `PackageManager` instance in `onCreate()`, caching it in a `mgr` data member. `getPackageList()` then calls `getInstalledPackages()` on `PackageManager`, specifically requesting to retrieve signature information via the `GET_SIGNATURES` flag. The list we get back from

`getInstalledPackages()` can be in any order, so we sort the results before returning it for display purposes:

```
  @Override
  public List<PackageInfo> getPackageList() {
    List<PackageInfo> result=
        mgr.getInstalledPackages(PackageManager.GET_SIGNATURES);

    Collections.sort(result, new Comparator<PackageInfo>() {
      @Override
      public int compare(final PackageInfo a, final PackageInfo b) {
        return(a.packageName.compareTo(b.packageName));
      }
    });

    return(result);
  }
```

(from MiscSecurity/SigDump/app/src/main/java/com/commonsware/android/signature/dump/MainActivity.java)

Note that this is a `List` of `PackageInfo` objects, so we need an `ArrayAdapter` subclass to handle rendering that. Here, we have a `PackageListAdapter` that knows how to populate list rows using the `packageName` field of a `PackageInfo` object, plus using an activated row layout for API Level 11+ devices:

```
package com.commonsware.android.signature.dump;

import android.annotation.TargetApi;
import android.content.pm.PackageInfo;
import android.os.Build;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;

class PackageListAdapter extends ArrayAdapter<PackageInfo> {
  PackageListAdapter(PackagesFragment packagesFragment) {
    super(packagesFragment.getActivity(), getRowResourceId(),
          packagesFragment.getContract().getPackageList());
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View result=super.getView(position, convertView, parent);

    ((TextView)result).setText(getItem(position).packageName);
```

```
    return(result);
  }

  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  private static int getRowResourceId() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      return(android.R.layout.simple_list_item_activated_1);
    }

    return(android.R.layout.simple_list_item_1);
  }
}
```

(from MiscSecurity/SigDump/app/src/main/java/com/commonsware/android/signature/dump/PackageListAdapter.java)

The result is that our master list is a list of all installed packages, sorted by package name, with the detail `TableLayout` peeking out of the right edge when shown on a phone-sized screen:

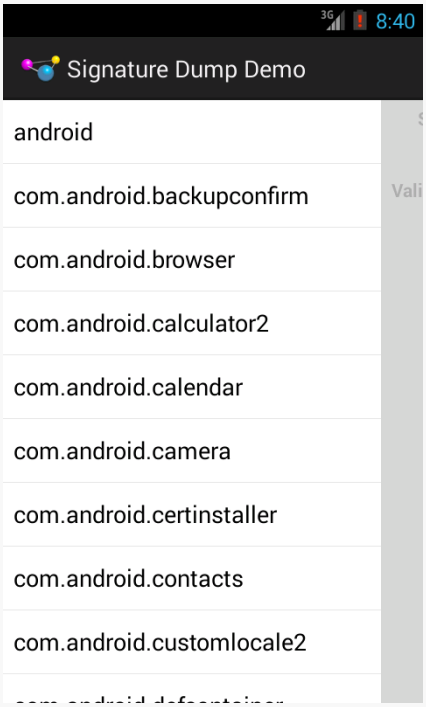


Figure 768: Signature Dump Demo, As Initially Launched

Dumping the Key

`onListItemClick()` of our `PackagesFragment` routes control to `onPackageSelected()` of the `Contract` interface, which in our case is `MainActivity`. There, we need to do some useful stuff based upon the fact that the user tapped on a particular package:

```
@Override
public void onPackageSelected(PackageInfo pkgInfo) {
  Signature[] signatures=pkgInfo.signatures;
  byte[] raw=signatures[0].toByteArray();

  sigDisplay.show(raw);
  panes.closePane();

  File output=
      new File(getExternalFilesDir(null),
               pkgInfo.packageName.replace('.', '_') + ".bin");

  new WriteThread(output, raw).start();
}
```

(from MiscSecurity/SigDump/app/src/main/java/com/commonsware/android/signature/dump/MainActivity.java)

First, we get the `Signature` array from the `PackageInfo` object. While this is an array, usually an app will only be signed once. Signing more than once is not especially useful, as an upgraded app needs to match the count and contents of each signature. Hence, we will only pay attention to the first signature. If you are using these techniques as the basis for your client JAR checking the public key of your app for IPC protection purposes, *and* your app is signed with multiple keys, you will want to check all of those keys.

The public key itself is represented as a byte array in the `Signature`. `onPackageSelected()` does two things with this byte array:

- Writes it to a file on external storage using a background thread, with a filename based on the app's package name, with `.` characters replaced by `_` characters
- Passes the byte array to the detail fragment (a `SignatureFragment`) and updates the `SlidingPaneLayout` to ensure that detail fragment is now visible

Decoding the Key

`SignatureFragment` is mostly comprised of the `show()` method that `MainActivity` uses to pass us the byte array of the "signature" to display:

```
package com.commonsware.android.signature.dump;

import android.app.Fragment;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import java.io.ByteArrayInputStream;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Locale;

public class SignatureFragment extends Fragment {
  DateFormat fmt=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.US);

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    return(inflater.inflate(R.layout.sig, container, false));
  }

  void show(byte[] raw) {
    CertificateFactory cf=null;

    try {
      cf=CertificateFactory.getInstance("X509");
    }
    catch (CertificateException e) {
      Log.e(getClass().getSimpleName(),
            "Exception getting CertificateFactory", e);
      return;
    }

    X509Certificate c=null;
    ByteArrayInputStream bin=new ByteArrayInputStream(raw);
```

```
    try {
      c=(X509Certificate)cf.generateCertificate(bin);
    }
    catch (CertificateException e) {
      Log.e(getClass().getSimpleName(),
            "Exception getting X509Certificate", e);
      return;
    }

    TextView tv=(TextView)getView().findViewById(R.id.subject);

    tv.setText(c.getSubjectDN().toString());

    tv=(TextView)getView().findViewById(R.id.issuer);
    tv.setText(c.getIssuerDN().toString());

    tv=(TextView)getView().findViewById(R.id.valid);
    tv.setText(fmt.format(c.getNotBefore()) + " to "
        + fmt.format(c.getNotAfter()));
  }
}
```

(from MiscSecurity/SigDump/app/src/main/java/com/commonsware/android/signature/dump/SignatureFragment.java)

That byte array *really* represents an X509 certificate, serialized to a byte array. `show()` goes through the work to get the `X509Certificate` object representing that same data, assuming the byte array is not corrupted somehow. Then, `show()` populates some `TextView` widgets in our `TableLayout` with the:

- The subject of the signature
- The issuer of the signature
- The range of dates in which this signature is valid

A debug signing key output will resemble:

Figure 769: Signature Dump Demo, Showing Debug Signing Key

A self-signed production signing key will resemble:

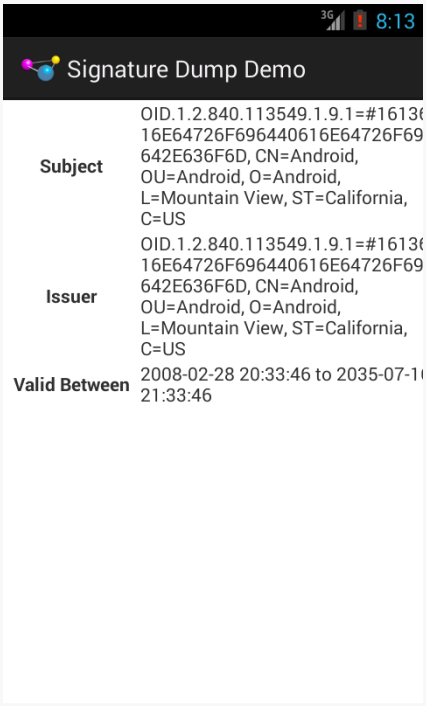


Figure 770: Signature Dump Demo, Showing Production Signing Key

A signing key created by some signing authority would have a subject that is distinct from its issuer.

Choosing Your Signing Keysize

The documentation for app signing contains a small side note about the `-keysize` parameter to `keytool`, the utility used to generate our signing keys:

> The size of each generated key (bits). If not supplied, Keytool uses a default key size of 1024 bits. In general, we recommend using a key size of 2048 bits or higher.

The reason for the 2,048-bit key size recommendation is that 1,024-bit RSA (the `keytool` default) has been considered at risk for a few years.

The recent revelations about state-sponsored decryption research should be hammering this home. Even if today, forging a 1,024-bit digital signature is still impractical for all but the largest security agencies, it is well within reason that this will fall within the reach of large botnets in the not-too-distant future. Once signing

keys can be cracked, apps will be able to be replaced with hacked editions, without tripping up the signature check, or `signature`-level permission checks might start passing due to forged signatures.

Switching to a larger keysize is not that hard... for new apps. Just specify `-keysize 4096` when creating your production signing key, and you should be good for a long time, barring a major decryption breakthrough for RSA signatures.

For existing apps with existing signing keys, though, you cannot change the key without breaking your ability to update the app.

Create a new, stronger production signing key, as a separate key from whatever you are using for production. Make note to use that new signing key for any new apps you create. And, if you have other reasons why you are migrating an existing user base to a new app (e.g., free app for which you are now offering a paid-app option), consider using the new signing key.

If you are a consultant, and you create unique signing keys per project, just cut over to using a stronger key for new clients and projects.

And if you are creating apps for which security is paramount, you might consider whether it is worthwhile to move your user base to a new version of the app with a new signing key at some point, just for the added protection.

Avoiding Accidental APIs

One place where developers create their own security problems is with "accidental APIs".

An API, of course, is where one code base exposes some interface that another code base can use. An accidental API is when one code base does not intend to expose an interface, but does anyway, possibly to the app's detriment.

Bear in mind that if your app becomes popular, other developers will poke and prod at it, to see if they can connect to your app by one means or another. Perhaps they want to offer features that you have not gotten to yet. Perhaps they have more nefarious aims. Regardless, making sure that other code can only work with your app the way that *you* intend for such code to work with your app.

Export Only What's Necessary

A component of your app is only reachable by a third-party app if it is exported. Otherwise, it is inaccessible to third-party apps.

(Admittedly, content providers have an exception to this rule, which we will get to shortly)

You normally do not think about exporting components, except when it comes to content providers. However, your choices for how you implement your app may lead you to accidentally export things that you did not realize were exported.

Export Defaults

The official way to declare whether or not a component is exported is to have an `android:exported` attribute for that component in the manifest (e.g., on an `<activity>` element). However, many times, we do not have such an attribute, but instead rely on the default export behavior.

Activities, services, and broadcast receivers have a simple rule for the default: if the component has an `<intent-filter>`, it is exported by default. Otherwise, it is not exported by default.

This, in turn, leads to a fairly simple development rule: only use an `<intent-filter>` and implicit `Intent` objects for working with your components if you *also* want third party apps to work with those components. Otherwise, do not use `<intent-filter>`, and instead communicate with your components using explicit `Intent` objects (e.g., the kind that take a Java class as the second constructor parameter).

For example, the classic `MAIN/LAUNCHER` `<intent-filter>` on your launcher activity is specifically there because you *want* a third party app — the launcher — to be able to start your activity. Most, if not all, of your other activities probably do not need an `<intent-filter>`, as they are likely to be private to your app.

The Chooser Bug

Some developers choose to still use an `<intent-filter>` and implicit `Intent` objects for their own private activities, yet then use `android:exported` to enforce the privacy.

This is not a good plan.

The rest of the system, notably `PackageManager`, does not pay much attention to `android:exported` *until the time when the component is to be used*, such as when the activity is to be started. Then, and only then, does Android realize that the component is not exported, and it fails the request, usually with a cryptic `SecurityException`.

A classic example of where this can cause problem came to light in 2012, with the UPS Mobile app. The rest of this section is an excerpt from the author's blog post on this incident:

The UPS Mobile app allows you to track packages and do a handful of other things that you might ordinarily do via the UPS Web site. It generally seems to be well-regarded, but it has an annoying flaw:

It claims to be Barcode Scanner, and does a lousy job at it.

Barcode Scanner, from ZXing, is a favorite among Android developers for its integration possibilities. However, some people do not like having a dependence upon the Barcode Scanner app, so they grab the open source code and attempt to blend it into their own apps. This is neither endorsed nor supported by the ZXing team, but since it is open source, it is also perfectly legitimate.

However, UPS (or whoever they hired to build the app) screwed up. They not only copied the source code, but they copied the manifest entry for the scanning activity. And, their activity has:

```
<intent-filter>
  <action android:name="com.google.zxing.client.android.SCAN" />
  <category android:name="android.intent.category.DEFAULT" />
<intent-filter>
```

This means that on any device that has UPS Mobile installed, they will be an option for handling Barcode Scanner `Intent` objects. What happened was that the person asking the question was manually invoking `startActivityForResult()` to bring up Barcode Scanner, was getting a chooser with UPS Mobile in it, and then was crashing upon choosing UPS Mobile... because UPS Mobile declared this activity to be not exported.

There was a bug, in which Android would display non-exported activities in a chooser, despite the fact that they could never be successfully used by the user. This appears to have been fixed as of Android 4.2, despite the issue being declined.

So, what should we learn from this?

First, UPS Mobile should not have used that `<intent-filter>`. As Dianne Hackborn has pointed out, your `<intent-filter>` mix is effectively part of your app's API, and so you need to think long and hard about every `<intent-filter>` you publish. UPS Mobile is not Barcode Scanner and should not be advertising that they handle such `Intent` objects, despite the activity being not exported.

Second, UPS Mobile probably should not have had *any* `<intent-filter>` elements for this activity, if they intend to use it purely internally. They could just as easily use an explicit `Intent` to identify the activity and avoid all of this nonsense.

Third, the person who filed the SO question ideally would have been using ZXing's `IntentIntegrator`. As Sean Owen of the ZXing project noted in a comment on my answer, `IntentIntegrator` ensures that only Barcode Scanner or official brethren will handle any scan requests, so this problem would not have appeared.

Fourth, Android really should not be showing non-exported activities in a chooser, which means probably that `PackageManager` should be filtering out non-exported activities from methods like `queryIntentActivities()`, which I presume lies at the heart of the chooser.

In summary, if your component is truly private, do not have an `<intent-filter>` on it, lest you cause yourself, and your users, problems with other apps.

The ContentProvider Behavior Change

Content providers are a little different... in lots of ways. In the specific scenarios being covered here, there are two primary differences.

First, third-party apps *can* still access a provider that has `android:exported="false"`. However, they can only do so in response to some operation initiated by your application, using `android:grantUriPermissions` and flags like `FLAG_GRANT_READ_URI_PERMISSION`. A third-party app will have no *independent* access to your non-exported provider.

Second, the default value for `android:exported` not only does not depend upon `<intent-filter>` (since few providers use one), but it has changed over the years:

- For apps with `android:minSdkVersion` *and* `android:targetSdkVersion` set to 16 or lower, the provider is exported by default
- All other apps, the provider is *not* exported by default

Lint will complain about your manifest having a `<provider>` without an `android:exported` attribute.

Sanitize Your Input Extras

If you do expose one or more of your components to third-party apps, and you are supporting certain `Intent` extras on any `Intent` objects used to talk to those components, make sure that the extras' values make sense.

Even Google makes this error, as was seen in the `PreferenceActivity` bug. `PreferenceActivity` supports an extra, named `:android:show_fragment`, to indicate that the activity should immediately jump to a specific fragment, rather than start at the top level of the preference navigation. The problem is that `PreferenceActivity` did not — and, at the time, could not — validate that the fragment to be loaded is a fragment that is *supposed* to be loaded. This would allow attackers to force apps, like Settings, to load arbitrary fragments, including those not normally accessible to the current user. This is the reason why we now need to override `isValidFragment()` in our `PreferenceActivity` implementations, so *we* can declare whether or not a particular requested fragment is a legitimate choice or not.

The equivalent behavior for a `ContentProvider` is to sanitize the inputs to methods like `query()`, `update()`, `openFile()`, and so on, to make sure that you do not expose something that you should not. For example, blindly accepting paths to `openFile()` could get you in trouble, if the `Uri` contains relative paths (e.g., `content://your.authority.here/../databases/your-private.db`), perhaps allowing third parties to get at files that you did not intend for them to access.

Secure Your Output Extras

Similarly, if you send broadcasts or otherwise use IPC to talk to third-party apps, bear in mind that others might be able to see some of that interaction, depending on the IPC in question.

The obvious case is with a broadcast `Intent` for an implicit `Intent`. Any app with a registered receiver will be able to "tune into" that broadcast and get whatever data is inside the `Intent`. In cases where you cannot use permissions to limit the scope of the broadcast, you need to make sure that there is nothing in the `Intent` that is private to the user.

Sometimes, though, non-obvious cases will emerge. For a few years, `Intent` extras on activities might be viewed by third-party apps that held the `GET_TASKS` permission, courtesy of the recent-tasks list. The `Intent` used to launch the task is available via `ActivityManager` and `getRecentTasks()`. While this specific problem was resolved in Android 4.1.1, there may be other similar scenarios lurking about.

Other Ways to Expose Data

Sometimes, we expose data to third-party apps by using standard Android APIs. We focus on the normal publisher and consumer of data using those APIs and forget about other apps that might be monitoring those communications. Or, we might not realize that one party in those communications may not have the user's best interests at heart. This section outlines some examples.

App Widgets

Any data that is put into the widgets inside of your `RemoteViews` for an app widget is visible to the home screen, lockscreen, or other app widget host. Those apps are the ones actually converting the `RemoteViews` into a view hierarchy, and they can inspect those views, reading the text in your `TextViews`, and so forth.

As a result, be careful about exposing potentially sensitive data via an app widget.

Notifications

Custom notifications also use `RemoteViews` and therefore could suffer from the same problem.

On the surface, you might not be worried quite so much about this, because the `Notification` object goes to the `NotificationManager`, for display by the OS itself.

However, as of Android 4.3 (API Level 18), apps can register to listen to added and removed notifications via a `NotificationListenerService`. Not only can such a service read the text from your `Notification`, but it can also access your

`RemoteViews`. This includes any `RemoteViews` that may be generated for you by the expanded notification classes (e.g., `BigPictureStyle`).

As a result, be careful about exposing potentially sensitive data via a `Notification`.

Clipboard

Any app can retrieve text off of the clipboard. After all, that's the point behind a clipboard.

However, this does mean that you need to be careful what you put on the clipboard in the first place. The quintessential problem case is a password manager: putting a password on the clipboard for easy pasting into an app's `EditText` password field will be popular, but it allows that password to be retrieved by other apps.

You can attempt to help reduce the window of risk by clearing the clipboard after a period of time. However, bear in mind that your process might be terminated before that occurs. Also, only clear the clipboard if the clipboard text is still yours — do not clear the clipboard if another app has already put its own contents there, lest you confuse and irritate the user in the middle of some other paste operation.

ServerSocket and Kin

If you open up any sort of server-style socket connection — TCP/IP, Bluetooth, etc. — bear in mind that the Android security framework may not be able to help you much. You cannot secure a `ServerSocket` with an `android:permission` attribute, for example. It is up to you to validate whether a particular request is expected and allowed, or not.

Jacking Attacks

Jacking attacks, in general, refer to cases where what the user *thinks* they are interacting with on-screen is not *actually* what they are interacting with. Instead, something else has interposed itself between the user and the activity that the user is trying to use. That "something else" might be trying to intercept user input (tapjacking, activity jacking) or confuse the user about what is actually being interacted with (window jacking).

Classic Tapjacking

Tapjacking refers to another program intercepting and inspecting touch events that are delivered to your foreground activity (or related artifacts, such as the input method editor). At its worst, tapjackers could intercept passwords, PINs, and other private data.

The term "tapjacking" seems to have been coined by Lookout Mobile Security, in a blog post that originally demonstrated this issue.

The Problem

You may recall that there are three axes to consider with Android user interfaces. The X and Y axes are the ones you typically think about, as they control the horizontal and vertical positioning of widgets in an activity. The Z axis — effectively "coming out the screen towards the user's eyes" — can be used in applications for sophisticated techniques, such as a pop-up panel.

Normally, you think of the Z axis within the scope of your activity and its widgets. However, there are ways to display "system alerts" – widgets that can float over the top of any activity. A `Toast` is the one you are familiar with, most likely. A `Toast` displays something on the screen, yet touch events on the `Toast` itself will be passed through to the underlying activity. Lookout demonstrated that it is possible to create a fully-transparent `Toast`. However, the lifetime of a `Toast` is limited (3.5 seconds maximum), which would limit how long it can try to grab touch events.

However, any application holding the `SYSTEM_ALERT_WINDOW` permission can display their own "system alerts" with custom look and custom duration. By making one that is fully transparent and lives as long as possible, a tapjacker can obtain touch events for any application in the system, including lock screens, home screens, and any standard activity.

On the surface, this might not seem terribly useful, since the `View` cannot see what is being tapped upon.

However, a savvy malware author would identify what activity is in the foreground and log that information along with the tap details and the screen size, periodically dumping that information to some server. The malware author can then scan the touch event dumps to see what interesting applications are showing up. With a minor investment – and possibly collaboration with other malware authors — the

author can know what touch events correspond to what keys on various input method editors, including the stock keyboards used by a variety of devices. Loading a pirated version of the APK on an emulator can indicate which activity has the password, PIN, or other secure data. Then, it is merely a matter of identifying the touch events applied to that activity and matching them up with the soft keyboard to determine what the user has entered. Over time, the malware author can perhaps develop a script to help automate this conversion.

Hence, the on-device tapjacker does not have to be very sophisticated, other than trying to avoid detection by the user. All of the real work to leverage the intercepted touch events can be handled offline.

How to Address This

In principle, Android 4.0.3 fixed this, by preventing touch events from being delivered to two separate applications. Either the tapjacking `View` gets the touch event (and consumes it), or the tapjacking `View` does not get the touch event (and therefore does not know about it).

For Android 2.2 and 2.3 devices, you also have the option of `setFilterTouchesWhenObscured()`, which will be examined later in this chapter.

Activity Jacking

In August 2014, a number of media outlets reported on a research paper and USENIX conference presentation describing a way by which your users could be tricked into providing confidential information — passwords, credit card information, and such — to a piece of malware, rather than to your app. This flew in the face of conventional wisdom, which said that the tapjacking fixes from Android 4.0.3 cleared up this sort of problem.

The paper points out that there are ways of writing malware such that:

- the malware can pop an *activity* in front of yours, and
- do so at just the right time, to mimic one of your activities, such that the user thinks that the malware's activity is actually yours and enters the confidential data into the malware activity

The authors describe it as a UI inference attack; to keep with the theme of this chapter, this section refers to it as "activity jacking".

The Problem

The details of how to execute the attack are rather esoteric, using lots of curious approaches to find out when an activity comes onto the screen and, more specifically, *which* activity of an app being attacked it is. Readers are encouraged to review the paper if you want details of exactly how to execute this sort of attack their way.

However, one simpler way of knowing this stuff is to implement an `AccessibilityService`. Officially, such services are supposed to help with accessibility, such as providing TalkBack-style audio announcements as the user navigates the UI by touch alone. In practice, a lot of apps use `AccessibilityService` to be able to monitor user inputs across the device and, in some cases, modify those inputs. Some password managers, for example, implement an `AccessibilityService` to help them auto-fill login dialogs. As a result, many users install and enable an `AccessibilityService` without really thinking about whether they can trust that service.

Given that you know when a particular activity appears on the screen, the attack is simple: launch your own activity that looks much like the original. The user might miss the fact that two activities just appeared, then go ahead and interact with your activity, thinking that it is from the real app. For example, you might interpose your own authentication dialog in front of the one for the banking app, thereby getting the user's PIN or passcode.

You can further take steps to try to "cover your tracks" and deal with the fact that the real activity is waiting for user input:

- If you are using an `AccessibiltyService`, you can use `performGlobalAction()` to initiate a BACK button press, right after dismissing your own activity, to dismiss the original activity.
- Otherwise, you can pretend that the user input is flawed and needs to be re-entered. In the case of an authentication dialog, you can pop up a regular `AlertDialog` that says that their password was not recognized. When the user dismisses that dialog, you also `finish()` your intercepting activity, returning the user to the real activity, where they can complete the real authentication.

How to Address This

An activity jack attack has two key weaknesses:

1. The attacker cannot see the screen, because on non-buggy devices, the attacker has no means of silently capturing a screenshot of our activity as it comes into the foreground. Hence, while the attacker can create an activity that tries to mimic ours, they can only do so statically, analyzing our activity's UI on their development machine and creating their own lookalike.
2. We know that our activity has left the foreground, as we are called with `onPause()` (and perhaps other lifecycle methods, depending upon the nature of the attacker).

Hence, one defense can be to include in our activity a **secure element** that cannot be mimicked ahead of time, then **hide that element** (or our whole UI) when we are no longer in the foreground.

This concept of a secure element is not new. Some financial services Web sites have taken this approach. As part of the user setting up their online banking account, the user chooses an image from a collection of clipart. On the Web page that collects the user's passphrase, the page also shows this secure element. The user is taught that if they do not see their chosen image, then the Web page they are looking at is not really from their bank, and therefore they should not type in their passphrase.

This is not that hard to implement in Android. You too would allow the user to choose a piece of clipart, displaying that in an `ImageView` on your secure activity in `onResume()`. In `onPause()` you would hide that `ImageView` via `setVisibility(View.INVISIBLE)`. That way:

- Since the image is chosen by the user, the attacker is unlikely to mimic the same image
- Since you are hiding the image when you are not in the foreground, the attacker cannot use a transparent region in their activity to have your image "peek through" their attacking activity

As a result, if the user is paying attention, the user should see either the wrong image or no image at all, and the user should realize that they are being activity jacked and therefore fail to proceed.

You might be tempted to do something else in response to your secure activity being replaced in the foreground by another app's activity, such as pop up a warning

dialog. However, there are plenty of valid scenarios when this would occur, such as an incoming phone call, and you have no reliable means of whitelisting all possible valid scenarios. There will be a high incidence of false positives, and that may not help the user. Having this as an user-selectable option is fine, but I would not go this route by default.

Window Jacking

Sometimes, the objective of the attacker is not to prevent the user from entering in information, or even to see what the user enters. Sometimes, the objective is to confuse the user, tricking them into clicking on things that they might not want to click on.

The Problem

A great example of this comes from Android 6.0's runtime permission system.

Apps with `targetSdkVersion` of 23 or higher will need to call `requestPermissions()` at various points, to ask the user to grant runtime permissions not previously granted (or granted but later revoked). That brings up a system-supplied dialog-themed activity:

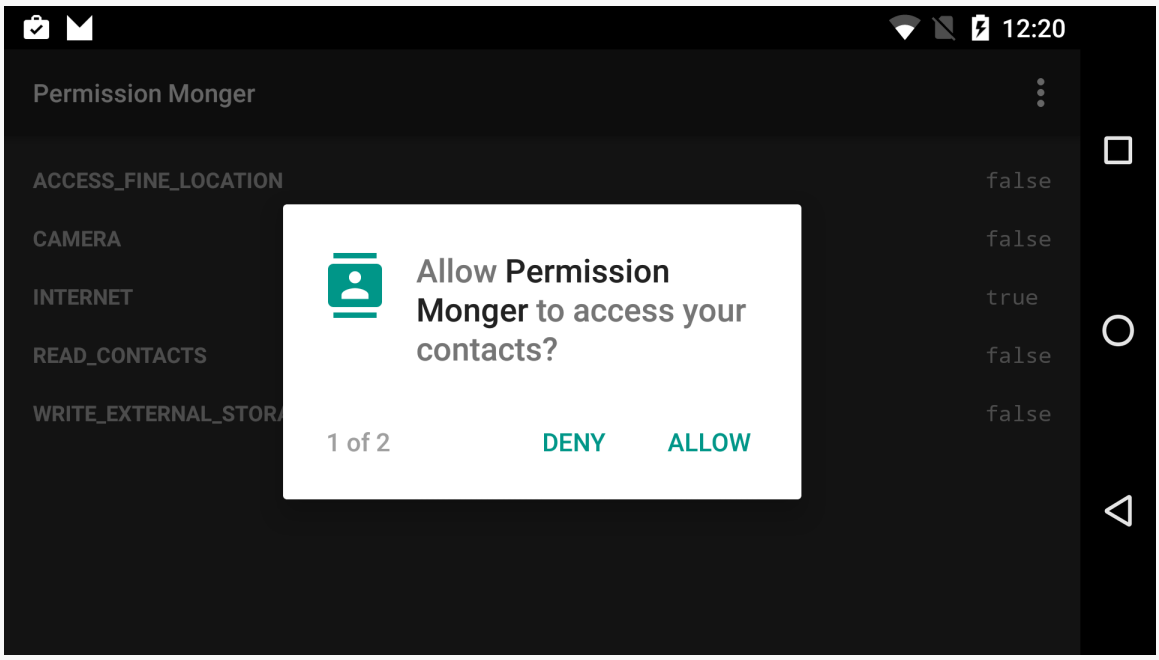


Figure 771: Runtime Permission System Dialog

Perhaps the attacker wants the user to agree to the permission but fears that the user might deny it instead. The attacker could use `SYSTEM_ALERT_WINDOW` to put a `View` *on top of the system dialog*, replacing the real permission explanation with something seemingly benign. The user — who may not have a lot of Android experience – clicks "Allow", where if the user were presented with the *real* message, the user might have clicked "Deny".

How to Address This

Quoting the Android documentation:

> Sometimes it is essential that an application be able to verify that an action is being performed with the full knowledge and consent of the user, such as granting a permission request, making a purchase or clicking on an advertisement. Unfortunately, a malicious application could try to spoof the user into performing these actions, unaware, by concealing the intended purpose of the view. As a remedy, the framework offers a touch filtering mechanism that can be used to improve the security of views that provide access to sensitive functionality.
>
> To enable touch filtering, call `setFilterTouchesWhenObscured(boolean)` or set the `android:filterTouchesWhenObscured` layout attribute to true. When enabled, the framework will discard touches that are received whenever the view's window is obscured by another visible window. As a result, the view will not receive touches whenever a toast, dialog or other window appears above the view's window.

For the runtime permission window jacking, using `setFilterTouchesWhenObscured()` would prevent the user from clicking on either the "Allow" or the "Deny" buttons. The alternative message would be in its own window, floating over the dialog. Hence, that should cause `FLAG_WINDOW_IS_OBSCURED` to be set on any `MotionEvents` delivered to the dialog, and those touch events would be dropped.

For example, take a look at the `res/layout/main.xml` file in the `Tapjacking/RelativeSecure` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
```

```
  android:layout_height="wrap_content"
  android:filterTouchesWhenObscured="true">
  <TextView android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="URL:"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"/>
  <EditText
    android:id="@id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/label"
    android:layout_alignParentTop="true"/>
  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
    android:layout_alignRight="@id/entry"
    android:text="OK" />
  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Cancel" />
</RelativeLayout>
```

(from Tapjacking/RelativeSecure/app/src/main/res/layout/main.xml)

Here, we have `android:filterTouchesWhenObscured="true"` on the `RelativeLayout` at the root of the layout resource. This property cascades to a container's children, and so if a tapjacker (or Toast or whatever) is above any of the widgets in the `RelativeLayout`, none of the touch events will be processed.

More fine-grained control can be achieved in custom widgets by overriding `onFilterTouchEventForSecurity()`, which gets control before the regular touch event methods. You can determine if a touch event had been intercepted by looking for the `FLAG_WINDOW_IS_OBSCURED` flag in the `MotionEvent` passed to `onFilterTouchEventForSecurity()`, and you can make the decision of how to handle this on an event-by-event basis.

The Problem with the Solution

According to Iwo Banaś, this approach may not actually work due to bugs in Android's implementation. The filter-when-obscured logic depends upon a `FLAG_WINDOW_IS_OBSCURED` value being on the `MotionEvent`, and that may be getting lost somewhere along the way.

The author of this book has not yet attempted to replicate Mr. Banaś' findings.

Google's Line of Defense: Obscuring the Foreground

Google's focus, besides the fixes listed above, is to make it increasingly difficult for one app to find out when another app is in the foreground. This is a key component of jacking attacks, as the jacker needs to know what is behind it. For example, with window jacking, obscuring the permission message only makes sense when the permission dialog appears — having some floating message appear at other points in time will be a giveaway that something is amiss.

As a result, methods on `ActivityManager` that used to provide details of all running processes have been neutered, frequently only providing details about your own process. Similarly, in Android 7.0, attempts by apps to find out about other processes through Linux-isms, like `/proc`, are being locked down.

Using FLAG_SECURE

By default, your activity's UI contents can be captured for any number of things:

- the overview screen (a.k.a., recent-tasks list)
- screenshots and screencasts, whether via the media projection APIs or some other device-supplied means
- the Assist API, such as Google's "Now On Tap" feature

However, you may have some activities that should not be captured in this fashion, due to potential privacy issues.

For that, you can apply `FLAG_SECURE` to an `Activity`:

```
public class FlagSecureTestActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    getWindow().setFlags(LayoutParams.FLAG_SECURE,
                         LayoutParams.FLAG_SECURE);

    setContentView(R.layout.main);
  }
}
```

Call `setFlags()` before `setContentView()`, in this case setting `FLAG_SECURE`.

In theory, this will prevent any of the aforementioned capture options from working.

Unfortunatately, the Android framework sometimes creates its own `Window` instances, such as the drop-down in a `Spinner`. Even if you set `FLAG_SECURE` on the `Window` for an activity, the Android framework does not pass that flag to any other windows created on behalf of that activity, and those windows show up in:

- Screenshots and screencasts taken by the media projection APIs on Android 5.0+
- The Assist API (e.g., Now On Tap) on Android 6.0+
- Android Studio screen recordings on Android 4.4+

This has been demonstrated to affect:

- `AutoCompleteTextView`
- `Spinner` (both dropdown and dialog modes)
- the overflow menu of the framework-supplied action bar
- `ShareActionProvider`
- `Dialog` and subclasses (e.g., `AlertDialog`)
- `Toast`

Of these, only the `Dialog` offers us access to its `Window`, on which we could apply `FLAG_SECURE`, for developers that realize that this is required.

Google has officially stated that all of this is working as intended.

If you are using `FLAG_SECURE`, you should thoroughly exercise your app's UI on Android 4.4+ while recording a screencast — the Android Studio screen recorder would be a simple tool to use. Then, play back that screencast, see what windows show up, and identify those that contain sensitive information that should not appear. Some of the windows that appear will not contain sensitive information —

here, the risk is that you might add sensitive information to them in the future but forget about this bug.

Then, you have two main courses of action: rewrite your UI to avoid the UI elements that are leaking this information, or attempt to patch the problem.

Trail: Data Storage and Retrieval

Content Provider Theory

Android publishes data to you via an abstraction known as a "content provider". Access to contacts and the call log, for example, are given to you via a set of content providers. In a few places, Android expects you to supply a content provider, such as for integrating your own search suggestions with the Android Quick Search Box. And, content providers are one way for you to supply data to third party applications, or to consume information from third party applications. As such, content providers have the potential to be something you would encounter frequently, even if in practice they do not seem used much.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on working with local databases.

Using a Content Provider

Any `Uri` in Android that begins with the `content://` scheme represents a resource served up by a content provider. Content providers offer data encapsulation using `Uri` instances as handles – you neither know nor care where the data represented by the `Uri` comes from, so long as it is available to you when needed. The data could be stored in a SQLite database, or in flat files, or retrieved off a device, or be stored on some far-off server accessed over the Internet.

Given a `Uri`, you may be able to perform basic CRUD (create, read, update, delete) operations using a content provider. `Uri` instances can represent either collections or individual pieces of content. Given a collection `Uri`, you may be able to create new pieces of content via insert operations. Given an instance `Uri`, you may be able to

read data represented by the `Uri`, update that data, or delete the instance outright. Or, given a `Uri`, you may be able to open up a handle to what amounts to a file, that you can read and, possibly, write to.

These are all phrased as "may" because the content provider system is a facade. The actual implementation of a content provider dictates what you can and cannot do, and not all content providers will support all capabilities.

Pieces of a Uri

A `Uri` for a `ContentProvider` is made up of two to four components.

A provider `Uri` always has a `content` scheme. So, when represented as a string, you will see the `Uri` start with `content://`.

After the scheme, where in an `http://` URL you would find a domain name or IP address, a provider `Uri` always has the authority string. This is unique on the device — only one provider will be tied to a given authority string.

What comes after the authority string is up to the provider. It is structured like the path segments of an `http://` URL, but what those path segments mean is up to the provider implementation. The one approximate rule is that a `Uri` pointing to an individual piece of content — such as a row of a table or view in a database – frequently has the `Uri` end in a number, where the number indicates a unique identifier of that content.

Most of the Android APIs expect these to be `Uri` objects, though in common discussion, it is simpler to think of them as strings. The `Uri.parse()` static method creates a `Uri` out of the string representation.

Getting a Handle

So, where do these `Uri` instances come from?

Some `Uri` values are part of the framework. For example, `ContactsContract.Contacts.CONTENT_URI` is a `Uri` pointing at the collection of contacts.

You might also get `Uri` instances handed to you from other sources, such as getting `Uri` handles for contacts via activities responding to `ACTION_PICK` or `ACTION_GET_CONTENT` `Intent` objects.

You can also hard-wire literal `String` objects (e.g., `"content://contacts/people"`) and convert them into `Uri` instances via `Uri.parse()`. This is not an ideal solution, as the base `Uri` values could conceivably change over time. For example, while you *used* to access contacts via a `Uri` like `content://contacts/people`, that is no longer the case. `ContactsContract.Contacts.CONTENT_URI` is a different value and will give you better results.

The Database-Style API

Of the two flavors of API that a content provider may support, the database-style API is more prevalent. Using a `ContentResolver`, you can perform standard "CRUD" operations (create, read, update, delete) using what looks like a SQL interface.

Makin' Queries

Given a base `Uri`, you can run a query to return data out of the content provider related to that `Uri`. This has much of the feel of SQL: you specify the "columns" to return, the constraints to determine which "rows" to return, a sort order, etc. The difference is that this request is being made of a content provider, not directly of some database (e.g., SQLite).

You have two main options for running a query:

1. Use the `query()` method on `ContentResolver` from some sort of background thread
2. Use a `CursorLoader`, as is discussed in an upcoming chapter

The standard `query()` method on `ContentResolver` takes five parameters:

- The base `Uri` of the content provider to query, or the instance `Uri` of a specific object to query
- An array of properties (think "columns") from that content provider that you want returned by the query
- A constraint statement, functioning like a SQL `WHERE` clause
- An optional set of parameters to bind into the constraint clause, replacing any `?` that appear there
- An optional sort statement, functioning like a SQL `ORDER BY` clause

This method returns a `Cursor` object, which you can use to retrieve the data returned by the query.

This will hopefully make more sense given an example. This chapter shows some sample bits of code from the ContentProvider/ConstantsPlus sample project. This is the same basic application as was first shown back in the chapter on database access, but rewritten to pull the database logic into a content provider, which is then used by a retained `ListFragment`.

As before, in `onViewCreated()`, we kick off a `LoadCursorTask` if we do not already have our `Cursor`, such as via a configuration change:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  SimpleCursorAdapter adapter=
      new SimpleCursorAdapter(getActivity(), R.layout.row,
          current, new String[] {
          DatabaseHelper.TITLE,
          DatabaseHelper.VALUE },
          new int[] { R.id.title, R.id.value },
          0);

  setListAdapter(adapter);

  if (current==null) {
    task=new LoadCursorTask(getActivity()).execute();
  }
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

`LoadCursorTask` inherits from a `BaseTask`. `BaseTask` and its subclasses need a `ContentResolver` to be able to work with our `ContentProvider`. So, `BaseTask` takes a `Context` in its constructor and uses that to retrieve a `ContentResolver`:

```
abstract private class BaseTask<T> extends AsyncTask<T, Void, Cursor> {
  final ContentResolver resolver;

  BaseTask(Context ctxt) {
    super();

    resolver=ctxt.getContentResolver();
  }
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

In `doInBackground()`, `LoadCursorTask` calls a `doQuery()` method inherited from `BaseTask`, which in turn uses our `ContentResolver` to query our `ContentProvider`:

```
  protected Cursor doQuery() {
    Cursor result=resolver.query(Provider.Constants.CONTENT_URI,
        PROJECTION, null, null, null);

    result.getCount();

    return(result);
  }
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

In the call to `query()`, we provide:

1. The `Uri` for our provider (`Provider.Constants.CONTENT_URI`), in this case representing the collection of physical constants managed by the provider
2. A list of properties to retrieve
3. Three `null` values, indicating that we do not need a constraint clause (the `Uri` represents the instance we need), nor parameters for the constraint, nor a sort order (we should only get one entry back)

The biggest "magic" here is the list of properties. The lineup of what properties are possible for a given provider should be provided by the documentation (or source code) for the content provider itself. In this case, we define logical values on the `Provider` provider implementation class that represent the various properties (namely, the unique identifier, the display name or title, and the value of the constant), and we refer to them with our `PROJECTION`:

```
private static final String[] PROJECTION=new String[] {
    Provider.Constants._ID, Provider.Constants.TITLE,
    Provider.Constants.VALUE };
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

Adapting to the Circumstances

Now that we have a `Cursor` via `query()`, we have access to the query results and can do whatever we want with them. You might, for example, manually extract data from the `Cursor` to populate widgets or other objects.

In our case, we are using the `SimpleCursorAdapter`, set up in `onViewCreated()`, to render our `Cursor`. This means that we need to take the `Cursor` that `doQuery()`

generates and arrange to hand that to the `SimpleCursorAdapter`. The `onPostExecute()` method on `BaseTask` handles this:

```
  @Override
  public void onPostExecute(Cursor result) {
    ((CursorAdapter)getListAdapter()).changeCursor(result);
    current=result;
    task=null;
  }
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

Give and Take

Of course, content providers would be astonishingly weak if you couldn't add or remove data from them, and were instead limited to only update what is there. Fortunately, content providers offer these abilities as well.

To insert data into a content provider, you have two options available on the `ContentProvider` interface (available through `getContentResolver()` to your activity):

- Use `insert()` with a collection `Uri` and a `ContentValues` structure describing the initial set of data to put in the row
- Use `bulkInsert()` with a collection `Uri` and an array of `ContentValues` structures to populate several rows at once

The `insert()` method returns a `Uri` for you to use for future operations on that new object. The `bulkInsert()` method returns the number of created rows; you would need to do a query to get back at the data you just inserted.

For example, if the user chooses our "Add" overflow item, we pop up a dialog to collect a new constant:

```
private void add() {
  LayoutInflater inflater=getActivity().getLayoutInflater();
  View addView=inflater.inflate(R.layout.add_edit, null);
  AlertDialog.Builder builder=new AlertDialog.Builder(getActivity());

  builder.setTitle(R.string.add_title).setView(addView)
      .setPositiveButton(R.string.ok, this)
      .setNegativeButton(R.string.cancel, null).show();
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

Then, if the user taps the "OK" button in the dialog, our `onClick()` listener is called, where we collect the entered values from the user, pour them into a `ContentValues` structure, and pass that to an `InsertTask`:

```
@Override
public void onClick(DialogInterface dialog, int which) {
  ContentValues values=new ContentValues(2);
  AlertDialog dlg=(AlertDialog)dialog;
  EditText title=(EditText)dlg.findViewById(R.id.title);
  EditText value=(EditText)dlg.findViewById(R.id.value);

  values.put(DatabaseHelper.TITLE, title.getText().toString());
  values.put(DatabaseHelper.VALUE, value.getText().toString());

  task=new InsertTask(getActivity()).execute(values);
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

`InsertTask`, in its `doInBackground()` method, calls `insert()` on a `ContentResolver` to insert this row:

```
  @Override
  protected Cursor doInBackground(ContentValues... values) {
    resolver.insert(Provider.Constants.CONTENT_URI, values[0]);

    return(doQuery());
  }
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

Notice that we also call `doQuery()` again. That is because our `Cursor` is now out of date, and we need to obtain a fresh `Cursor` with fresh results. And, as with `LoadTask`, `InsertTask` inherits from `BaseTask`, not only providing us with that `doQuery()` method but also the `onPostExecute()` method that puts the `Cursor` into the `SimpleCursorAdapter`.

To delete one or more rows from the content provider, use the `delete()` method on `ContentResolver`. This works akin to a SQL `DELETE` statement and takes three parameters:

- A `Uri` representing the collection (or instance) from which you wish to delete rows

- A constraint statement, functioning like a SQL `WHERE` clause, to determine which rows should be deleted
- An optional set of parameters to bind into the constraint clause, replacing any `?` that appear there

The Streaming API

Sometimes, what you are trying to retrieve does not look like a set of rows and columns, but rather looks like a stream. For example, the `MediaStore` provider manages the index of all music, video, and image files available on external storage, and you can use `MediaStore` to open up a stream to read in the contents of one of those files. Here, working with the `Uri` and the provider is much like working with a URL and a Web server.

Some content providers, like `MediaStore`, support both the database-style and streaming APIs — you query to find media that matches your criteria, then can open some file that matches. Other content providers might only support the streaming API.

Working with the Stream

Given a `Uri` that represents some file managed by the content provider, you can use `openInputStream()` and `openOutputStream()` on a `ContentResolver` to access an `InputStream` or `OutputStream`, respectively. Note, though, that not all content providers may support both modes. For example, a content provider that serves files stored inside the application (e.g., assets in the APK file), you will not be able to get an `OutputStream` to modify the content.

Also note that `openInputStream()` and `openOutputStream()` work with both `file://` and `content://` `Uri` values — you do not need to manually inspect the `Uri` and handle files separately if you do not want to.

Retrieving Metadata

You can call `getType()` on a `ContentResolver`, supplying a `Uri` as a parameter. This will return the MIME type reported by the `ContentProvider` for the data at that `Uri`. For the streaming API, this will give you results reminiscent of a Web server — some specific MIME type if the provider knows it, otherwise probably some generic MIME type (e.g., `application/octet-stream`).

You can also call `query()` on the `ContentResolver`. Your projection (the list of columns to return) can include:

- `OpenableColumns.SIZE`, which will return the length of the file being streamed to you for that `Uri`, and
- `OpenableColumns.DISPLAY_NAME`, which should be some name for the file that the user might recognize

The DATA Anti-Pattern

However, the authors of `MediaStore` screwed up developer expectations, due to a legacy convention.

The legacy convention was that a `content://` `Uri` might not be openable directly using something like `openInputStream()`. Instead, it pointed to a database row, retrievable via `query()`, and you would look in the `DATA` column for how to access the actual data. Some providers no doubt continue to use this pattern, as does `MediaStore`. The rules for what the `DATA` column would be were not well documented, but by convention they tended to be a path to a file. The problem is that this runs afoul of Google's current guidance, as there is no guarantee that other apps can access such a file.

Do not blindly assume that if you get a `content://` `Uri` that it is for the `DATA` pattern. Try to open a stream on the `Uri`, and if that fails, then see if the `DATA` pattern is in play. Or, if you `query()` to get the size and/or display name first, also request the `DATA` column, and if it exists and is not `null`, try that if opening the stream directly does not work.

Building Content Providers

Building a content provider is a very tedious task. There are many requirements of a content provider, in terms of methods to implement and public data members to supply. And, until you try using it, you have no great way of telling if you did any of it correctly (versus, say, building an activity and getting validation errors from the resource compiler).

That being said, building a content provider is of huge importance if your application wishes to make data available to other applications. If your application is keeping its data solely to itself, you may be able to avoid creating a content provider, just accessing the data directly from your activities. But, if you want your data to

possibly be used by others — for example, you are building a feed reader and you want other programs to be able to access the feeds you are downloading and caching — then a content provider is right for you.

First, Some Dissection

The content `Uri` is the linchpin behind accessing data inside a content provider. When using a content provider, all you really need to know is the provider's base `Uri`; from there you can run queries as needed, or construct a `Uri` to a specific instance if you know the instance identifier.

When building a content provider, though, you need to know a bit more about the innards of the content `Uri`.

A content `Uri` has two to four pieces, depending on situation:

1. It always has a scheme (`content://`), indicating it is a content `Uri` instead of a `Uri` to a Web resource (`http://`).
2. It always has an authority, which is the first path segment after the scheme. The authority is a unique string identifying the content provider that handles the content associated with this `Uri`.
3. It may have a data type path, which is the list of path segments after the authority and before the instance identifier (if any). The data type path can be empty, if the content provider only handles one type of content. It can be a single path segment (`foo`) or a chain of path segments (`foo/bar/goo`) as needed to handle whatever data access scenarios the content provider requires.
4. It may have an instance identifier, which is an integer identifying a specific piece of content. A content `Uri` without an instance identifier refers to the collection of content represented by the authority (and, where provided, the data path).

For example, a content `Uri` could be as simple as `content://sekrits`, which would refer to the collection of content held by whatever content provider was tied to the `sekrits` authority (e.g., `SecretsProvider`). Or, it could be as complex as `content://sekrits/card/pin/17`, which would refer to a piece of content (identified as `17`) managed by the `sekrits` content provider that is of the data type `card/pin`.

Next, Some Typing

Next, you need to come up with some MIME types corresponding with the content your content provider will provide. There are three basic patterns.

For the streaming API, the MIME type that you will use should be the actual MIME type of the stream itself. Perhaps you already know the MIME type (e.g., you got it in an HTTP header when you downloaded the content from a Web server). Perhaps you will use `MimeTypeMap` to try to infer a MIME type based on a file extension. That is up to you, just as it is up to you to ensure that your Web server returns proper MIME types for streams that it serves up.

For the database-style API, even though the MIME type system is not really designed for this sort of thing, we still use MIME types. Each `Uri` will have an associated MIME type, indicating what is represented by that `Uri`. A `Uri` that points to a collection of content (e.g., a database table or view) will use one MIME type structure, while a `Uri` that points to an individual piece of content (e.g., a row in that database table or view) will use a different MIME type structure.

The collection MIME type should be of the form `vnd.X.cursor.dir/Y`, where `X` is the name of your firm, organization, or project, and `Y` is a dot-delimited type name. So, for example, you might use `vnd.tlagency.cursor.dir/sekrits.card.pin` as the MIME type for your collection of secrets.

The instance MIME type, for an individual piece of content, should be of the form `vnd.X.cursor.item/Y`, usually for the same values of `X` and `Y` as you used for the collection MIME type (though that is not strictly required).

Implementing the Database-Style API

Just as an activity and a receiver are both Java classes, so is a content provider. So, the big step in creating a content provider is crafting its Java class, with a base class of `ContentProvider`.

In your subclass of `ContentProvider`, you are responsible for implementing six methods that, when combined, perform the services that a content provider is supposed to offer to activities wishing to create, read, update, or delete content via the database-style API.

Implement onCreate()

As with an activity, the main entry point to a content provider is `onCreate()`. Here, you can do whatever initialization you want. In particular, here is where you should lazy-initialize your data store. For example, if you plan on storing your data in such-and-so directory on external storage, with an XML file serving as a "table of contents", you should check and see if that directory and XML file are there and, if not, create them so the rest of your content provider knows they are out there and available for use.

Similarly, if you have rewritten your content provider sufficiently to cause the data store to shift structure, you should check to see what structure you have now and adjust it if what you have is out of date.

Implement query()

As one might expect, the `query()` method is where your content provider gets details on a query some activity wants to perform. It is up to you to actually process said query.

The query method gets, as parameters:

1. A `Uri` representing the collection or instance being queried
2. A `String` array representing the list of properties that should be returned
3. A `String` representing what amounts to a SQL `WHERE` clause, constraining which instances should be considered for the query results
4. A `String` array representing values to "pour into" the `WHERE` clause, replacing any `?` found there
5. A `String` representing what amounts to a SQL `ORDER BY` clause

You are responsible for interpreting these parameters however they make sense and returning a `Cursor` that can be used to iterate over and access the data.

As you can imagine, these parameters are aimed towards people using a SQLite database for storage. You are welcome to ignore some of these parameters (e.g., you elect not to try to roll your own SQL `WHERE` clause parser), but you need to document that fact so activities only attempt to query you by instance `Uri` and not by using parameters that you elect to ignore.

Implement insert()

Your `insert()` method will receive a `Uri` representing the collection and a `ContentValues` structure with the initial data for the new instance. You are responsible for creating the new instance, filling in the supplied data, and returning a `Uri` to the new instance.

Implement update()

Your `update()` method gets the `Uri` of the instance or collection to change, a `ContentValues` structure with the new values to apply, a String for a SQL `WHERE` clause, and a `String` array with parameters to use to replace `?` found in the `WHERE` clause. Your responsibility is to identify the instance(s) to be modified (based on the `Uri` and `WHERE` clause), then replace those instances' current property values with the ones supplied.

This will be annoying, unless you are using SQLite for storage. Then, you can pretty much pass all the parameters you received to the `update()` call to the database, though the `update()` call will vary slightly depending on whether you are updating one instance or several.

Implement delete()

As with `update()`, `delete()` receives a `Uri` representing the instance or collection to work with and a `WHERE` clause and parameters. If the activity is deleting a single instance, the `Uri` should represent that instance and the `WHERE` clause may be null. But, the activity might be requesting to delete an open-ended set of instances, using the `WHERE` clause to constrain which ones to delete.

As with `update()`, though, this is simple if you are using SQLite for database storage (sense a theme?). You can let it handle the idiosyncrasies of parsing and applying the `WHERE` clause — all you have to do is call `delete()` on the database.

Implement getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

Update the Manifest

The glue tying the content provider implementation to the rest of your application resides in your `AndroidManifest.xml` file. Simply add a `<provider>` element as a child of the `<application>` element, such as:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="18"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <provider
      android:name=".Provider"
      android:authorities="com.commonsware.android.constants.Provider"
      android:exported="false"/>

    <activity
      android:name=".ConstantsBrowser"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from ContentProvider/ConstantsPlus/app/src/main/AndroidManifest.xml)

The `android:name` property is the name of the content provider class, with a leading dot to indicate it is in the stock namespace for this application's classes (just like you use with activities).

The `android:authorities` property should be a semicolon-delimited list of the authority values supported by the content provider. Recall, from earlier in this chapter, that each content `Uri` is made up of a scheme, authority, data type path, and instance identifier. Each authority from each `CONTENT_URI` value should be included in the `android:authorities` list. Now, when Android encounters a content `Uri`, it can sift through the providers registered through manifests to find a matching authority. That tells Android which application and class implements the content provider, and from there Android can bridge between the calling activity and the content provider being called.

Several other attributes relate to security:

- `android:exported` indicates whether third-party apps are able to initiate communications with your provider on their own
- `android:readPermission` and `android:writePermission` allow you to defend your provider with permissions; third-party apps have to have `<uses-permission>` elements for those permissions to be able to work with your provider
- `android:grantUriPermissions` indicates whether you are able to selectively "poke pinholes in the firewall" of your provider security, to say that for specific IPC operations (e.g., starting a third-party activity), that third party has limited access to your provider's content

These will be explored later in this book.

Add Notify-On-Change Support

A feature that your content provider can offer to its clients is notify-on-change support. This means that your content provider will let clients know if the data for a given content `Uri` changes.

For example, suppose you have created a content provider that retrieves RSS and Atom feeds from the Internet based on the user's feed subscriptions (via OPML, perhaps). The content provider offers read-only access to the contents of the feeds, with an eye towards several applications on the phone using those feeds versus everyone implementing their own feed poll-fetch-and-cache system. You have also implemented a service that will get updates to those feeds asynchronously, updating

the underlying data store. Your content provider could alert applications using the feeds that such-and-so feed was updated, so applications using that specific feed can refresh and get the latest data.

On the content provider side, to do this, call `notifyChange()` on your `ContentResolver` instance (available in your content provider via `getContext().getContentResolver()`). This takes two parameters: the `Uri` of the piece of content that changed and the `ContentObserver` that initiated the change. In many cases, the latter will be `null`; a non-`null` value simply means that the observer that initiated the change will not be notified of its own changes.

On the content consumer side, an activity can call `registerContentObserver()` on its `ContentResolver` (via `getContentResolver()`). This ties a `ContentObserver` instance to a supplied `Uri` — the observer will be notified whenever `notifyChange()` is called for that specific `Uri`. When the consumer is done with the `Uri`, `unregisterContentObserver()` releases the connection.

Implementing the Streaming API

If you want to have a `ContentProvider` support streaming data via the streaming API, you will still need to set up the `<provider>` element, choose an authority, and create a subclass of `ContentProvider` as with the database-style API. From there, whether you are adding the streaming API to an existing provider or creating a new one, there is some additional work to be done.

Serving the Stream

If you want consumers of your `ContentProvider` to be able to call `openInputStream()` or `openOutputStream()` on a `Uri`, the most likely approach is to implement the `openFile()` method. The `openFile()` method returns a curious object called a `ParcelFileDescriptor`. Given that, the `ContentResolver` can obtain the `InputStream` or `OutputStream` that was requested. There are various static methods on `ParcelFileDescriptor` to create instances of it, such as an `open()` method that takes a `File` object as the first parameter. Note that this works for both files on external storage and files within your own project's app-local file storage (e.g., `getFilesDir()`).

`openFile()` also gets a `String` parameter that is the "mode" for opening the file. This can be converted into appropriate flags for use with `ParceFileDescriptor` and its

`open()` method. Mostly, this is for determining whether we are opening the file for read or write operations.

Serving the Metadata

You should implement the `query()` method in your provider as well. If the `Uri` is pointing to one of your streams, you should create a one-row `MatrixCursor` and supply the `OpenableColumns` as the columns. `OpenableColumns` has two values: `DISPLAY_NAME` (for some human-readable name of the stream) and `SIZE` (the length of the stream in bytes). Based on the `projection` string array passed into `query()`, you can skip columns that the client is not requesting.

You also need to implement `getType()`. For the database-style API, you pretty much invent your own MIME types. For the streaming API, you should be returning MIME types for the `Uri` values that really represent the contents of that `Uri`. In other words, your `getType()` method should behave like you would expect a Web server to do with respect to the `Content-Type` header. If you know the MIME type for certain (e.g., you got it yourself in an HTTP or IMAP operation and saved it), use that. If you do not know the MIME type for certain, you can try the `MimeTypeMap` class, which knows how to map common file extensions to their MIME type counterparts. Worst-case, return `application/octet-stream`.

The Rest of the Requirements

You also have to implement the following `abstract` methods:

- `onCreate()`
- `insert()`
- `update()`
- `delete()`

If you are not supporting the database-style API, you are welcome to have `insert()`, `update()`, and `delete()` throw some `RuntimeException`, to indicate that those operations are not supported.

Issues with Content Providers

Content providers are not without their issues.

The biggest complaint seems to be the lack of an `onDestroy()` companion to the `onCreate()` method you can implement. Hence, if you open a database in `onCreate()`, you close it... never. Sometimes, you can alleviate this by initializing things on demand and releasing them immediately, such as opening a database as part of `insert()` and closing it within the same method. This does not always work, however — for example, you cannot close the database you query in `query()`, since the `Cursor` you return would become invalid. Holding onto an open `SQLiteDatabase` is not a problem, as all of your data changes are written to disk as part of committing transactions. So, many `ContentProvider` implementations settle for simply never closing the database.

The fact that `ContentProvider` is effectively a facade means that a consumer of a `ContentProvider` has no idea what to expect. It is up to documentation to explain what `Uri` values can be used, what columns can be returned, what query syntax is supported, and so on. And, the fact that it is a facade means that much of the richness of the SQLite interface is lost, such as `GROUP BY`. To top it off, the API supported by `ContentProvider` is rather limited — if what you want to share does not look like a database and does not look like a file, it may be difficult to force it into the `ContentProvider` API.

Another issue is the client's dependence upon the provider itself. If, for whatever reason, the provider's process is terminated while the client has an open `Cursor` on query results, the client's process is also terminated. It is unclear if the same effect occurs when the client has an open stream from a provider through the streaming API, though it seems likely. Now, in theory, the importance of the provider's process should be raised to the highest importance of any of its clients, though this behavior is not documented and may not occur in practice.

This behavior by Android is rather drastic, more drastic than what happens to HTTP clients when the Web server they are connected to crashes. There, the client winds up with some sort of exception and can move on. The moral of this story is: when working with a `ContentProvider`, it behooves you to use the data quickly, particularly if your app is in the background at the time.

Content Provider Implementation Patterns

The previous chapter focused on the concepts, classes, and methods behind content providers. This chapter more closely examines some implementations of content providers, organized into simple patterns.

Prerequisites

Understanding this chapter requires that you have read the preceding chapter, along with the chapter on permissions.

The Single-Table Database-Backed Content Provider

The simplest database-backed content provider is one that only attempts to expose a single table's worth of data to consumers. The `CallLog` content provider works this way, for example.

Step #1: Create a Provider Class

We start off with a custom subclass of `ContentProvider`, named, cunningly enough, Provider. Here we need the database-style API methods: `query()`, `insert()`, `update()`, `delete()`, and `getType()`.

onCreate()

Here is the `onCreate()` method for `Provider`, from the ContentProvider/ConstantsPlus sample application:

```
  @Override
  public boolean onCreate() {
    db=new DatabaseHelper(getContext());

    return(true);
  }
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

While that does not seem all that special, the "magic" is in the private `DatabaseHelper` object, a fairly conventional `SQLiteOpenHelper` implementation:

```
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.hardware.SensorManager;

class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";
  static final String TITLE="title";
  static final String VALUE="value";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    Cursor c=db.rawQuery("SELECT name FROM sqlite_master WHERE type='table' AND name='constants'", null);

    try {
      if (c.getCount()==0) {
        db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, value
REAL);");

        ContentValues cv=new ContentValues();

        cv.put(Provider.Constants.TITLE, "Gravity, Death Star I");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Earth");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_EARTH);
        db.insert("constants", Provider.Constants.TITLE, cv);
```

```
      cv.put(Provider.Constants.TITLE, "Gravity, Jupiter");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_JUPITER);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Mars");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MARS);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Mercury");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MERCURY);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Moon");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MOON);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Neptune");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_NEPTUNE);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Pluto");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_PLUTO);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Saturn");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SATURN);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Sun");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SUN);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, The Island");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_THE_ISLAND);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Uranus");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_URANUS);
      db.insert("constants", Provider.Constants.TITLE, cv);

      cv.put(Provider.Constants.TITLE, "Gravity, Venus");
      cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_VENUS);
      db.insert("constants", Provider.Constants.TITLE, cv);
    }
  }
  finally {
    c.close();
  }
 }

 @Override
 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  android.util.Log.w("Constants", "Upgrading database, which will destroy all old data");
  db.execSQL("DROP TABLE IF EXISTS constants");
  onCreate(db);
 }
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

Note that we are creating the `DatabaseHelper` in `onCreate()` and are never closing it. That is because there is no `onDestroy()` (or equivalent) method in a `ContentProvider`. While we might be tempted to open and close the database on every operation, that will not work, as we cannot close the database and still hand back a live `Cursor` from the database. Hence, we leave it open and assume that SQLite's transactional nature will ensure that our database is not corrupted when Android shuts down the `ContentProvider`.

query()

For SQLite-backed storage providers like this one, the `query()` method implementation should be largely boilerplate. Use a `SQLiteQueryBuilder` to convert the various parameters into a single SQL statement, then use `query()` on the builder to actually invoke the query and give you a `Cursor` back. The `Cursor` is what your `query()` method then returns.

For example, here is `query()` from `Provider`:

```
@Override
public Cursor query(Uri url, String[] projection, String selection,
                    String[] selectionArgs, String sort) {
  SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

  qb.setTables(TABLE);

  String orderBy;

  if (TextUtils.isEmpty(sort)) {
    orderBy=Constants.DEFAULT_SORT_ORDER;
  }
  else {
    orderBy=sort;
  }

  Cursor c=
      qb.query(db.getReadableDatabase(), projection, selection,
               selectionArgs, null, null, orderBy);

  c.setNotificationUri(getContext().getContentResolver(), url);

  return(c);
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

We create a `SQLiteQueryBuilder` and pour the query details into the builder, notably the name of the table that we query against and the sort order (substituting in a default sort if the caller did not request one). When done, we use the `query()` method on the builder to get a `Cursor` for the results. We also tell the resulting `Cursor` what `Uri` was used to create it, for use with the content observer system.

insert()

Since this is a SQLite-backed content provider, once again, the implementation is mostly boilerplate: validate that all required values were supplied by the activity, merge your own notion of default values with the supplied data, and call `insert()` on the database to actually create the instance.

For example, here is `insert()` from `Provider`:

```
@Override
public Uri insert(Uri url, ContentValues initialValues) {
  long rowID=
      db.getWritableDatabase().insert(TABLE, Constants.TITLE,
                                      initialValues);

  if (rowID > 0) {
    Uri uri=
        ContentUris.withAppendedId(Provider.Constants.CONTENT_URI,
                                   rowID);
    getContext().getContentResolver().notifyChange(uri, null);

    return(uri);
  }

  throw new SQLException("Failed to insert row into " + url);
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

The pattern is the same as before: use the provider particulars plus the data to be inserted to actually do the insertion.

update()

Here is `update()` from `Provider`:

```
@Override
public int update(Uri url, ContentValues values, String where,
```

```
                  String[] whereArgs) {
  int count=
      db.getWritableDatabase()
        .update(TABLE, values, where, whereArgs);

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

In this case, updates are always applied across the entire collection, though we could have a smarter implementation that supported updating a single instance via an instance `Uri`.

delete()

Similarly, here is `delete()` from `Provider`:

```
@Override
public int delete(Uri url, String where, String[] whereArgs) {
  int count=db.getWritableDatabase().delete(TABLE, where, whereArgs);

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

This is almost a clone of the `update()` implementation described above.

getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

For example, here is `getType()` from `Provider`:

```
@Override
public String getType(Uri url) {
  if (isCollectionUri(url)) {
```

```
      return("vnd.android.cursor.dir/constant");
    }

    return("vnd.android.cursor.item/constant");
  }
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

Step #2: Supply a Uri

You may wish to add a public static member... somewhere, containing the `Uri` for each collection your content provider supports, for use by your own application code. Typically, this is a public static final `Uri` put on the content provider class itself:

```
  public static final Uri CONTENT_URI=
      Uri.parse("content://com.commonsware.android.constants.Provider/constants");
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

You may wish to use the same namespace for the content `Uri` that you use for your Java classes, to reduce the chance of collision with others.

Bear in mind that if you intend for third parties to access your content provider, they will not have access to this public static data member, as your class is not in their project. Hence, you will need to publish the string representation of this `Uri` that they can hard-wire into their application.

Step #3: Declare the "Columns"

Remember those "columns" you referenced when you were using a content provider, in the previous chapter? Well, you may wish to publish public static values for those too for your own content provider.

Specifically, you may want a public static class implementing `BaseColumns` that contains your available column names, such as this example from `Provider`:

```
public static final class Constants implements BaseColumns {
  public static final Uri CONTENT_URI=
      Uri.parse("content://com.commonsware.android.constants.Provider/constants");
  public static final String DEFAULT_SORT_ORDER="title";
  public static final String TITLE="title";
  public static final String VALUE="value";
}
```

(from ContentProvider/ConstantsPlus/app/src/main/java/com/commonsware/android/constants/Provider.java)

Since we are using SQLite as a data store, the values for the column name constants should be the corresponding column names in the table, so you can just pass the projection (array of columns) to SQLite on a `query()`, or pass the `ContentValues` on an `insert()` or `update()`.

Note that nothing in here stipulates the types of the properties. They could be strings, integers, or whatever. The biggest limitation is what a `Cursor` can provide access to via its property getters. The fact that there is nothing in code that enforces type safety means you should document the property types well, so people attempting to use your content provider know what they can expect.

Step #4: Update the Manifest

Finally, we need to add the provider to the `AndroidManifest.xml` file, by adding a `<provider>` element as a child of the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="18"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <provider
      android:name=".Provider"
      android:authorities="com.commonsware.android.constants.Provider"
      android:exported="false"/>

    <activity
      android:name=".ConstantsBrowser"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
```

```
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
 </application>

</manifest>
```

(from ContentProvider/ConstantsPlus/app/src/main/AndroidManifest.xml)

The Local-File Content Provider

Implementing a content provider that supports serving up files based on `Uri` values is similar, and generally simpler, than creating a content provider for the database-style API. In this section, we will examine the `ContentProvider/Files` sample project. This project demonstrates a common use of the filesystem-style API: serving files from internal storage to third-party applications (who, by default, cannot read your internally-stored files).

Note that this sample project will only work on devices that have an application capable of viewing PDF files accessed via `content://` `Uri` values.

The FileProvider Class

Our `ContentProvider` is named `FileProvider`. However, most of the logic is contained in an `AbstractFileProvider` that will be used for a handful of sample apps in this chapter. We will look at both of those classes, focusing first on the `FileProvider`.

onCreate()

We have an `onCreate()` method. In many cases, this would not be needed for this sort of provider. After all, there is no database to open. In this case, we use `onCreate()` to copy the file(s) out of assets into the app-local file store. In principle, this would allow our application code to modify these files as the user uses the app (versus the unmodifiable editions in `assets/`).

```
@Override
public boolean onCreate() {
  File f=new File(getContext().getFilesDir(), "test.pdf");

  if (!f.exists()) {
```

```
    AssetManager assets=getContext().getAssets();

    try {
      copy(assets.open("test.pdf"), f);
    }
    catch (IOException e) {
      Log.e("FileProvider", "Exception copying from assets", e);

      return(false);
    }
  }

  return(true);
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/FileProvider.java)

This uses a static `copy()` method, inherited from `AbstractFileProvider`, that can copy an `InputStream` from an asset to a local `File`. We will take a peek at this later in this chapter.

openFile()

We need to implement `openFile()`, to return a `ParcelFileDescriptor` corresponding to the supplied `Uri`:

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
  throws FileNotFoundException {
  File root=getContext().getFilesDir();
  File f=new File(root, uri.getPath()).getAbsoluteFile();

  if (!f.getPath().startsWith(root.getPath())) {
    throw new
      SecurityException("Resolved path jumped beyond root");
  }

  if (f.exists()) {
    return(ParcelFileDescriptor.open(f, parseMode(mode)));
  }

  throw new FileNotFoundException(uri.getPath());
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/FileProvider.java)

We are passed in a *nix-style string `mode`, which will be a value like `r` for read access, `wt` for write access (and truncate the file), etc. In API Level 19+, `ParcelFileDescriptor` has a convenience method for converting such modes into the equivalent `ParcelFileDescriptor` flag values. For older devices, you can simply use the `parseMode()` code that Google added:

```
// following is from ParcelFileDescriptor source code
// Copyright (C) 2006 The Android Open Source Project
// (even though this method was added much after 2006...)

private static int parseMode(String mode) {
  final int modeBits;
  if ("r".equals(mode)) {
    modeBits=ParcelFileDescriptor.MODE_READ_ONLY;
  }
  else if ("w".equals(mode) || "wt".equals(mode)) {
    modeBits=
        ParcelFileDescriptor.MODE_WRITE_ONLY
            | ParcelFileDescriptor.MODE_CREATE
            | ParcelFileDescriptor.MODE_TRUNCATE;
  }
  else if ("wa".equals(mode)) {
    modeBits=
        ParcelFileDescriptor.MODE_WRITE_ONLY
            | ParcelFileDescriptor.MODE_CREATE
            | ParcelFileDescriptor.MODE_APPEND;
  }
  else if ("rw".equals(mode)) {
    modeBits=
        ParcelFileDescriptor.MODE_READ_WRITE
            | ParcelFileDescriptor.MODE_CREATE;
  }
  else if ("rwt".equals(mode)) {
    modeBits=
        ParcelFileDescriptor.MODE_READ_WRITE
            | ParcelFileDescriptor.MODE_CREATE
            | ParcelFileDescriptor.MODE_TRUNCATE;
  }
  else {
    throw new IllegalArgumentException("Bad mode '" + mode + "'");
  }
  return modeBits;
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/FileProvider.java)

Our `openFile()` method then uses `parseMode()` in the call to the static `open()` method on `ParcelFileDescriptor`, which opens the file (with the desired access mode) and gives us our `ParcelFileDescriptor` back that we can return. If the file is not found, we can throw a `FileNotFoundException` to indicate that.

However, we also check to see that the `File` that we are trying to access is inside `getFilesDir()`, by comparing paths. A `Uri` can have `..` path segments to move up directory levels. Using that with the `File` constructor means that a rogue `Uri` could move outside of our designated root directory (`getFilesDir()`), to perhaps try to access other data on our internal storage (e.g., databases). `getAbsoluteFile()` will net out any path-traversal segments (e.g., `..`). If `getAbsoluteFile()` lies within `getFilesDir()`, we go ahead, otherwise we throw a `SecurityException`.

getDataLength()

`AbstractFileProvider` gives us a callback — `getDataLength()` — where we can indicate how big a file is, given its `Uri`. That information will be made available to clients consuming this stream. The default will be to indicate that the file size is unknown... and that usually works. However, if it is easy for you to determine the file size, do so, and it will increase the compatibility of your app with possible consumers.

In this case, determining the size of a local file is easy:

```
@Override
protected long getDataLength(Uri uri) {
  File f=new File(getContext().getFilesDir(), uri.getPath());

  return(f.length());
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/FileProvider.java)

The AbstractFileProvider Class

`AbstractFileProvider` is designed to handle a lot of common boilerplate for streaming providers like the one provided in this sample.

getType()

Just as our database-style `ContentProvider` needed to implement `getType()` to provide a MIME type given a `Uri`, so too do our streaming providers. The difference

is that a streaming provider usually wants to use "real" MIME types, values that third-party apps are likely to recognize. For example, a PDF file should use a MIME type of `application/pdf`, as that is what PDF viewing apps will expect.

Android has some convenience code for determining a likely MIME type. You can use `MimeTypeMap` to convert a file extension to a MIME type, or you can use `guessContentTypeFromName()` on`URLConnection` to get a MIME type for a URL. Both use the same underlying database — the difference is mostly a matter of whether you have a bare file extension already or not. So, the default implementation of `getType()` in `AbstractFileProvider` uses `guessContentTypeFromName()`:

```
@Override
public String getType(Uri uri) {
  return(URLConnection.guessContentTypeFromName(uri.toString()));
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/AbstractFileProvider.java)

If you know that *your* MIME type is unlikely to be recognized by Android (e.g., you invented your own), a subclass of `AbstractFileProvider` could handle those cases, chaining to the superclass for other `Uri` values.

insert(), update(), and delete()

`ContentProvider` itself is `abstract`, requiring us to implement a variety of methods to satisfy the compiler. Three of them — `insert()`, `update()`, and `delete()` — have no role in a pure-streaming `ContentProvider`, so `AbstractFileProvider` has stub implementations:

```
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
  throw new RuntimeException("Operation not supported");
}

@Override
public int update(Uri uri, ContentValues values, String where,
                  String[] whereArgs) {
  throw new RuntimeException("Operation not supported");
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
  throw new RuntimeException("Operation not supported");
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/AbstractFileProvider.java)

A `ContentProvider` that supports *both* the database-style and streaming APIs will need real implementations of those methods for the database operations, perhaps throwing an `Exception` for requests to insert, update, or delete a `Uri` that represents a stream.

query() and getFileName()

We also need to implement `query()`. You can get by with having this be a stub similar to `insert()` and kin. However, for better compatibility, you should have a more robust `query()` implementation, as it will be used by `ContentResolver` to retrieve two pieces of metadata about a `Uri`:

- What is a valid filename to use to represent this `Uri`, should we need a human-readable name? After all, a `ContentProvider` `Uri` does not have to represent a human-readable path, and so the last segment of that `Uri` could be a cryptic string of hex digits or something, not a filename.
- What is the length of the data that should be delivered by the stream?

`query()` will be called with a projection that contains either `OpenableColumns.DISPLAY_NAME`, `OpenableColumns.SIZE`, or both. A streaming `ContentProvider` ideally supports returning a `Cursor` with this data. The `AbstractFileProvider` implementation of `query()` handles this for us:

```
abstract class AbstractFileProvider extends ContentProvider {
  private final static String[] OPENABLE_PROJECTION= {
      OpenableColumns.DISPLAY_NAME, OpenableColumns.SIZE };

  @Override
  public Cursor query(Uri uri, String[] projection, String selection,
                      String[] selectionArgs, String sortOrder) {
    if (projection == null) {
      projection=OPENABLE_PROJECTION;
    }

    final MatrixCursor cursor=new MatrixCursor(projection, 1);

    MatrixCursor.RowBuilder b=cursor.newRow();

    for (String col : projection) {
      if (OpenableColumns.DISPLAY_NAME.equals(col)) {
        b.add(getFileName(uri));
      }
```

```
      else if (OpenableColumns.SIZE.equals(col)) {
        b.add(getDataLength(uri));
      }
      else { // unknown, so just add null
        b.add(null);
      }
    }

    return(new LegacyCompatCursorWrapper(cursor));
  }
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/AbstractFileProvider.java)

If the supplied `projection` is `null`, we assume that the caller wants the standard `OpenableColumns`; otherwise, we will use the supplied `projection`.

Our results will be packaged in a `MatrixCursor`. This amounts to a `Cursor` interface on a two-dimensional array, where you build up the rows in that array via a `MatrixCursor.RowBuilder`. In our case, there will only be one such row, for the relevant values for the file to be streamed in support of the requested `Uri`.

We iterate over the columns in the projection, calling out to `getFileName()` and `getDataLength()` methods for `OpenableColumns.DISPLAY_NAME` and `OpenableColumns.SIZE` respectively (and using `null` as the result for anything else). The default implementations of those methods return the last path segment of the `Uri` and `AssetFileDescriptor.UNKNOWN_LENGTH`, respectively:

```
  protected String getFileName(Uri uri) {
    return(uri.getLastPathSegment());
  }

  protected long getDataLength(Uri uri) {
    return(AssetFileDescriptor.UNKNOWN_LENGTH);
  }
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/AbstractFileProvider.java)

Subclasses can override those as needed, as we saw with `getDataLength()` in the concrete `FileProvider` class.

However, `query()` does not return the `MatrixCursor` directly. Instead, it wraps it in a `LegacyCompatCursorWrapper`. This class comes from the CWAC-Provider project, from the author of this book. `LegacyCompatCursorWrapper` is designed to try to improve compatibility with clients that are expecting `query()` results to include a `_DATA` column, the way that `MediaStore` does. Poorly-written clients will crash if this

column does not exist. `LegacyCompatCursorWrapper` wraps a `Cursor` and serves up an empty `_DATA` column for those clients that need one.

copy()

`AbstractFileProvider` also has a convenience `copy()` static method that copies an `InputStream` to a `File`, used from the `FileProvider` `onCreate()` method:

```
  static void copy(InputStream in, File dst)
                                                    throws IOException {
    FileOutputStream out=new FileOutputStream(dst);
    byte[] buf=new byte[1024];
    int len;

    while ((len=in.read(buf)) >= 0) {
      out.write(buf, 0, len);
    }

    in.close();
    out.close();
  }
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/AbstractFileProvider.java)

The Manifest

Finally, we need to add the provider to the `AndroidManifest.xml` file, by adding a `<provider>` element as a child of the `<application>` element, as with any other content provider:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.cp.files"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>
```

```
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="FilesCPDemo"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <provider
      android:name=".FileProvider"
      android:authorities="com.commonsware.android.cp.files"
      android:exported="true"/>
  </application>

</manifest>
```

(from ContentProvider/Files/app/src/main/AndroidManifest.xml)

Note, however, that we have `android:exported="true"` set in our `<provider>` element. This means that this content provider can be accessed from third-party apps or other external processes (e.g., the media framework for playing back videos).

Using this Provider

The activity is fairly trivial, simply creating an `ACTION_VIEW Intent` on our PDF file and starting up an activity for it, then finishing itself:

```
package com.commonsware.android.cp.files;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

public class FilesCPDemo extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
```

```
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(FileProvider.CONTENT_URI
                                 + "test.pdf")));
    finish();
  }
}
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/FilesCPDemo.java)

Here, we use a `CONTENT_URI` published by `FileProvider` as the basis for identifying the file:

```
  public static final Uri CONTENT_URI=
      Uri.parse("content://com.commonsware.android.cp.files/");
```

(from ContentProvider/Files/app/src/main/java/com/commonsware/android/cp/files/FileProvider.java)

The Protected Provider

The problem with the preceding example is that any app on the device, if it knows the right `Uri` to ask for, will be able to access the file. This may be desired, but often times it will not be. Instead, you may want to specifically indicate which apps, at specific points in time, can view the file.

Particularly if your objective is to start a third-party app to work with that file, setting up this sort of security is not that difficult. To see how that works, we will walk through the `ContentProvider/GrantUriPermissions` sample project. This is a clone of the `ContentProvider/Files` project with this extra security added on.

The way the defense works is by using Android's permission system. We will mark the `ContentProvider` as being not exported, then selectively grant that access to a specific `Uri` to the app that we want to view our file.

Step #1: Mark the Provider as Not Exported

Putting `android:exported="false"` on the `<provider>` element indicates that no app has the ability to make requests of your `ContentProvider`, except for specific cases where you authorize it:

```
    <provider
      android:name="FileProvider"
      android:authorities="com.commonsware.android.cp.files"
      android:exported="false"
```

```
      android:grantUriPermissions="false"
      tools:ignore="ExportedContentProvider">
      <grant-uri-permission android:path="/test.pdf"/>
    </provider>
```

(from ContentProvider/GrantUriPermissions/app/src/main/AndroidManifest.xml)

With no other changes, if we tried to use the app, the third-party PDF viewer would crash when trying to read our PDF file from the `Uri`.

Step #2: Grant Access to the Uri

To allow third parties to get access only when we specify, we need to make a few more changes.

This `<provider>` element also has `android:grantUriPermissions="false"`. That is the default value for this attribute, shown here purely for illustration purposes. It also has a `<grant-uri-permissions>` child element, listing the local path (within the `ContentProvider`) to our PDF file.

The `<grant-uri-permissions>` element (or elements, plural) allow us to override the permission requirement for certain pieces of content, granting access to that content on a per-request basis. There are three possibilities:

1. If `android:grantUriPermissions` is `true`, then we will be able to grant access to any content within our provider
2. If `android:grantUriPermissions` is `false`, but we have `<grant-uri-permissions>` sub-elements, we can only grant access to the content identified by the `Uri` paths specified in those sub-elements
3. If `android:grantUriPermissions` is `false`, and we have no `<grant-uri-permissions>` sub-elements (the default case), we cannot grant access to any content within our provider

In this case, we specify that we will only grant access to `/test.pdf`. Since that is the only content in this provider, we could have the same net effect by setting `android:grantUriPermissions` to `true`.

Then, when we create an `Intent` used to interact with another component, we can include a flag indicating what permission we wish to grant:

```
package com.commonsware.android.cp.perms;

import android.app.Activity;
```

```
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

public class FilesCPDemo extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    Intent i=new Intent(Intent.ACTION_VIEW, Uri.parse(FileProvider.CONTENT_URI + "test.pdf"));

    i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    startActivity(i);
    finish();
  }
}
```

(from ContentProvider/GrantUriPermissions/app/src/main/java/com/commonsware/android/cp/perms/FilesCPDemo.java)

In this revised version of our activity, we add `FLAG_GRANT_READ_URI_PERMISSION` to the `Intent` used with `startActivity()`. This will grant the activity that responds to our `Intent` read access to the specific `Uri` in the `Intent`, overriding the exported status. That is why, when you run this app on a device, the PDF viewer will still be able to view the file.

There is also `FLAG_GRANT_WRITE_URI_PERMISSION` for granting write access, not needed here, as our provider only supports read access.

While this is most commonly used with `startActivity()` (e.g., allowing a mail program limited access to your attachments provider), this can also be used with `startService()`, `bindService()`, and the various flavors of sending broadcasts (e.g., `sendBroadcast()`).

The Stream Provider

Sometimes, we want a provider that looks like the local-file provider from the preceding section... but we do not have a file. Instead, we have data in some other form, such as a byte array, or a `String`, or an `InputStream`. Writing that material to a file may be problematic, or even counterproductive.

For example, imagine an app that stores data on the user's behalf in an encrypted fashion. One such file is a PDF, that the user would like to view. There are PDF viewers that can view files served via `content://` `Uri` values, as the previous section demonstrated... but that assumes an unencrypted file. While we could decrypt the file, writing the decrypted results to another file, and serve the decrypted data to the PDF viewer, now we have a persistent decrypted version of the data. That opens a

window of time when the data might be accessed by people with nefarious intent, which is something we are trying to avoid by using the encrypted store in the first place. Rather, it would be nice if we could decrypt the data *on the fly* and give that decrypted result to the PDF viewer. Of course, there are security risks intrinsic to that too — after all, we do not know what the PDF viewer might do with the unencrypted data — but it is at least an improvement.

The good news is that Android does support streaming options for `openFile()`-style `ContentProvider` implementations. However, as one might expect, they are not the simplest things to implement.

In this section, we will examine the `ContentProvider/Pipe` sample project. This is a near clone of the `ContentProvider/Files` sample from the preceding section. However, rather than simply handing the file to Android to serve as content, we will stream it in ourselves. In principle, as part of this streaming, we could be decrypting it from an encrypted state. Since this sample shares much code with the previous sample, we will focus solely on the changes here.

Note that this sample was inspired by the sample found at https://github.com/nandeeshwar/Pfd-Create-Pipe.

The Pipes

Starting with API Level 9, it is possible to create a pipe between two processes, from the Android SDK, via `ParcelFileDescriptor`. In the previous section, we saw how `ParcelFileDescriptor` could be used to open a local file and make that available to other processes — the `createPipe()` method gives us a pipe.

The "pipe" returned by `createPipe()` is a two-element array of `ParcelFileDescriptor` objects. The first element in the array represents the "read" end of the pipe. In our case, that is the end that should be used by a PDF viewer to read in the file contents. The second element of the array represents the "write" end of the pipe, which we will use to supply the file's contents to the "read" end (and to the PDF viewer by extension).

The Revised openFile()

With that in mind, here is our revised `openFile()` method:

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
```

```
    throws FileNotFoundException {
    ParcelFileDescriptor[] pipe=null;

    try {
      pipe=ParcelFileDescriptor.createPipe();
      AssetManager assets=getContext().getAssets();

      new TransferThread(assets.open(uri.getLastPathSegment()),
                         new AutoCloseOutputStream(pipe[1])).start();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception opening pipe", e);
      throw new FileNotFoundException("Could not open pipe for: "
          + uri.toString());
    }

    return(pipe[0]);
  }
```

(from ContentProvider/Pipe/app/src/main/java/com/commonsware/android/cp/pipe/PipeProvider.java)

We create our pipe via `createPipe()`, then get an `InputStream` on our PDF file stored as an asset — unlike the `ContentProvider/Files` sample, we do not need to copy the asset to a local file now. We then kick off a background thread, implemented in an inner class named `TransferThread`, to actually copy the data from the asset to the write end of the pipe.

Rather than supply `TransferThread` with a `ParcelFileDescriptor` for the write end of the pipe, we supply an `OutputStream`. Specifically, we pass in a `ParcelFileDescriptor.AutoCloseOutputStream`. This is an `OutputStream` that knows to close the `ParcelFileDescriptor` when we close the stream. Otherwise, it behaves like a fairly typical `OutputStream`.

The Transfer

`TransferThread` is a fairly conventional copy-data-from-stream-to-stream implementation:

```
  static class TransferThread extends Thread {
    InputStream in;
    OutputStream out;

    TransferThread(InputStream in, OutputStream out) {
      this.in=in;
      this.out=out;
```

```
  }

  @Override
  public void run() {
    byte[] buf=new byte[1024];
    int len;

    try {
      while ((len=in.read(buf)) >= 0) {
        out.write(buf, 0, len);
      }

      in.close();
      out.flush();
      out.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
            "Exception transferring file", e);
    }
  }
}
```

(from ContentProvider/Pipe/app/src/main/java/com/commonsware/android/cp/pipe/PipeProvider.java)

Here, we read in data in 1KB blocks from the `InputStream` (our asset) and write the data to our `OutputStream` (obtained from the `ParcelFileDescriptor`).

The Results

Our activity logic has not substantially changed. We still create an `ACTION_VIEW` `Intent` on the `content://` `Uri` from our provider, pointing to our `test.pdf` asset. Any PDF viewer capable of handling `content://` `Uri` values will use a `ContentResolver` to open an `InputStream` for our `Uri`. In the `ContentProvider/Files` sample, that `InputStream` would receive the contents of the file directly from Android. In this new sample, that `InputStream` is reading in bytes off of our pipe, until such time as it has read in all the streamed data and we have closed the `OutputStream`.

Not every possible consumer of a `Uri` will be able to work with our stream, though. For example, `MediaPlayer` expects to be able to move forwards and backwards within the stream, and while that works for file-backed `ParcelFileDescriptors`, it does not work for those representing a pipe. Hence, `MediaPlayer` will crash when trying to use a `Uri` to a pipe-based stream, which is certainly unfortunate.

The author would like to thank Reuben Scratton for his assistance in tracking down this `MediaPlayer` limitation.

FileProvider

The Android Support package now contains its own implementation of a `FileProvider` that greatly simplifies serving files from internal or external storage to another app.

Here, we will see Google's `FileProvider` in action via the `ContentProvider/V4FileProvider` sample project. This is a near clone of the `ContentProvider/Pipe` sample from the preceding section, just leveraging `FileProvider` to help us serve a file from internal storage.

The Rationale

The documentation for `FileProvider` states:

> Apps should generally avoid sending raw filesystem paths across process boundaries, since the receiving app may not have the same access as the sender. Instead, apps should send Uri backed by a provider like FileProvider.

This is not just an issue for passing files from internal storage to other apps. On Android 4.2+ tablets, it could even be an issue for external storage, as each user account gets its own portion of external storage. There may be scenarios in which your app (associated with one user) winds up needing to pass the contents of a file on external storage to another app (associated with another user). Regular filesystem paths will not work in this case, as one user account cannot directly access another user account's files, even on external storage.

The Sources of Files

Google's `FileProvider` offers automatic serving of files from a few root points:

- `getFilesDir()` (i.e., the standard portion of internal storage for your app)
- `getCacheDir()` (i.e., internal storage, but files that the OS can purge if needed to free up disk space)
- `Environment.getExternalStorageDirectory()` (i.e., the root of external storage)

- `getExternalFilesDir(null)` and `getExternalCacheDir()` (i.e., unique directories on external storage for your app)

For each of these, you will be able to specify a specific subdirectory's worth of files that should be served, if you do not want the entire directory's contents published via `FileProvider`. You will also be able to specify an alias, which serves as the first path segment (after the authority in the `content://` `Uri`) — `FileProvider` maps that path segment to a specific location of files to serve.

The Manifest Entry

The information about what files to serve comes in the form of an XML resource file. You can name the file whatever you like, but its content needs to be a root `<paths>` element, with a series of children for the different directories you wish to serve. Those directories will be denoted via child elements with specific names:

- `<files-path>` for `getFilesDir()`
- `<cache-path>` for `getCacheDir()`
- `<external-path>` for `Environment.getExternalStorageDirectory()`
- `<external-files-path>` for `getExternalFilesDir(null)`
- `<external-cache-path>` for `getExternalCachePath()`

Note that the latter two require version 24.2.0 or higher of the support library, as they are fairly new.

For example, our sample project has a `res/xml/provider_paths.xml` file with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
    <files-path name="stuff" />
</paths>
```

(from ContentProvider/V4FileProvider/app/src/main/res/xml/provider_paths.xml)

Here, we are saying that we want to serve the contents of `getFilesDir()`, using a virtual root path of `stuff`. With an authority of `com.commonsware.android.cp.v4file`, this means that a `Uri` of `content://com.commonsware.android.cp.v4file/stuff/test.pdf` would serve up a `test.pdf` file in the `getFilesDir()` directory.

The optional `path` attribute of the `<files-path>`, etc. elements indicates a particular subdirectory, relative to the element-specific root, that should be used as the source of files. So, for example, had the `provider_paths.xml` file looked like:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <files-path name="stuff" path="help/" />
</paths>
```

...then `content://com.commonsware.android.cp.v4file/stuff/test.pdf` would map to `help/test.pdf` inside of `getFilesDir()`.

You then point to this XML resource from a `<meta-data>` element in the `<provider>` element in the manifest, teaching `FileProvider` what to serve. For example, our `<provider>` element in this sample app is:

```
    <provider
      android:name="LegacyCompatFileProvider"
      android:authorities="com.commonsware.android.cp.v4file"
      android:exported="false"
      android:grantUriPermissions="true">
      <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths"/>
    </provider>
```

(from ContentProvider/V4FileProvider/app/src/main/AndroidManifest.xml)

Here, our `android:name` points to a `LegacyCompatFileProvider` class that we will examine shortly. We still provide the `android:authorities` value, along with any permission rules that we want. Beyond that, we have a `<meta-data>` element, with an `android:name` of `android.support.FILE_PROVIDER_PATHS`, that points to our XML resource with the path information.

You will also notice that our `android:exported` attribute is set to `false`. As it turns out, `FLAG_GRANT_READ_URI_PERMISSION` trumps the exported status of a provider. If you pass a `Uri` to an activity using `FLAG_GRANT_READ_URI_PERMISSION`, the activity will be able to read the contents of that `Uri`, even if the provider itself is not exported.

The Legacy Compatibility

`LegacyCompatFileProvider` is a simple subclass of `FileProvider`, one that overrides `query()` and wraps its `Cursor` in a `LegacyCompatCursorWrapper` to try to improve compability with ill-behaved clients:

```
package com.commonsware.android.cp.v4file;

import android.database.Cursor;
import android.net.Uri;
import android.support.v4.content.FileProvider;
import com.commonsware.cwac.provider.LegacyCompatCursorWrapper;

public class LegacyCompatFileProvider extends FileProvider {
  @Override
  public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String
sortOrder) {
    return(new LegacyCompatCursorWrapper(super.query(uri, projection, selection, selectionArgs,
sortOrder)));
  }
}
```

(from ContentProvider/V4FileProvider/app/src/main/java/com/commonsware/android/cp/v4file/LegacyCompatFileProvider.java)

The Usage

At this point, the provider is ready for use, insofar as we can specify `Uri` values like `content://com.commonsware.android.cp.v4file/stuff/test.pdf` and get results. Of course, we actually need to have files in our internal storage, and we need to use such a `Uri`.

Hence, our activity combines the unpack-the-file-from-assets logic from our own providers in earlier samples, plus starts up a PDF viewer on our designated `test.pdf` file:

```
package com.commonsware.android.cp.v4file;

import android.app.Activity;
import android.content.Intent;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.support.v4.content.FileProvider;
import android.util.Log;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
```

```
public class FilesCPDemo extends Activity {
  private static final String AUTHORITY="com.commonsware.android.cp.v4file";

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    File f=new File(getFilesDir(), "test.pdf");

    if (!f.exists()) {
      AssetManager assets=getAssets();

      try {
        copy(assets.open("test.pdf"), f);
      }
      catch (IOException e) {
        Log.e("FileProvider", "Exception copying from assets", e);
      }
    }

    Intent i=
        new Intent(Intent.ACTION_VIEW,
                   FileProvider.getUriForFile(this, AUTHORITY, f));

    i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

    startActivity(i);
    finish();
  }

  static private void copy(InputStream in, File dst) throws IOException {
    FileOutputStream out=new FileOutputStream(dst);
    byte[] buf=new byte[1024];
    int len;

    while ((len=in.read(buf)) > 0) {
      out.write(buf, 0, len);
    }

    in.close();
    out.close();
  }
}
```

(from ContentProvider/V4FileProvider/app/src/main/java/com/commonsware/android/cp/v4file/FilesCPDemo.java)

`FileProvider` offers a handy `getUriForFile()` static helper method that will return a `Uri` for a given file, incorporating our specified content provider authority.

The result of running this activity is the same as the other file-serving provider samples from this chapter: a PDF viewer (if one is available) will display the `test.pdf` file.

StreamProvider

`FileProvider` is rather nice: you can serve up typical file-based content without having to roll your own implementation of `ContentProvider` and `openFile()`. However, it only supports a few sources of data.

The author of this book has written `StreamProvider`, a fork of `FileProvider` that adds support for serving content from assets and raw resources. Plus, through subclassing, you can readily serve up content from other sources as well. `StreamProvider` can be found in the CWAC-Provider project.

You can add this library to your Android Studio project much in the same way as you can other CWAC libraries: add the CWAC repository and request the dependency:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'com.commonsware.cwac:provider:0.5.0'
}
```

Once you have added the CWAC-Provider dependency to your project, you use it much the same as you would use `FileProvider`:

- Define an XML metadata file with a `<paths>` root element, containing one or more elements describing what you want the provider to serve
- Add `com.commonsware.cwac.provider.StreamProvider` as a `<provider>` to your manifest, under your own `android:authority`, with a `<meta-data>` element (with a name of `com.commonsware.cwac.provider.STREAM_PROVIDER_PATHS`), pointing to that XML metadata

```
<provider
  android:name="com.commonsware.cwac.provider.StreamProvider"
```

```
  android:authorities="..."
  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data
    android:name="com.commonsware.cwac.provider.STREAM_PROVIDER_PATHS"
    android:resource="@xml/..."/>
  <meta-data
    android:name="com.commonsware.cwac.provider.USE_LEGACY_CURSOR_WRAPPER"
    android:value="true"/>
</provider>
```

- Consider adding the `USE_LEGACY_CURSOR_WRAPPER` `<meta-data>` element, shown in the above example, to automatically add in `LegacyCompatCursorWrapper` support, described elsewhere in this chapter
- Use `FLAG_GRANT_READ_URI_PERMISSION` and `FLAG_GRANT_WRITE_URI_PERMISSION` in `Intent` objects you use to have third parties use the files the `StreamProvider` serves, to allow those apps selective, temporary access to the file

Exporting and Usage Patterns

If your `StreamProvider` is exported, all of your streams will be considered read-only, regardless of any other configuration. Mostly, this mode is here for cases where you need a streaming provider and cannot grant `Uri` permissions (e.g., implementing a `ChooserTargetService`).

If your `StreamProvider` is not exported, and it has `android:grantUriPermissions` set, then you can control, on a per-`Uri` basis, which clients get access to your streams. This works identically to how `FileProvider` works. Whether a particular source of streams is read-only or read-write will depend on whether the stream is a file and your metadata configuration.

Wherever possible, elect to not export the provider and use `FLAG_GRANT_READ_URI_PERMISSIONS` or similar techniques to selectively grant access to your content.

Note that the exported-and-read-only rule is on a per-provider basis. If you have some content that needs to be published globally and others that are not:

- Use `StreamProvider` and one `<provider>` element for one set of content, with one authority and `android:exported` setting

- Subclass `StreamProvider` and have a separate `<provider>` element for the other set of content, with a separate authority and `android:exported` setting

Metadata Elements

Google's `FileProvider` supports:

- `<files-path>` for serving files from your app's `getFilesDir()`
- `<external-path>` for serving files from `Environment.getExternalStoragePublicDirectory()`
- `<cache-path>` for serving files from your app's `getCacheDir()`
- `<external-files-path>` for serving files from `getExternalFilesDir()`
- `<external-cache-path>` for serving files from `getExternalCacheDir()`

Each of those take a `name` attribute, indicating the first path segment of the `Uri` that should identify this particular source of files. For example, a `name` of `foo` would mean that `content://your.authority.here/foo/...` would look for a `...` file in that particular element's source of files.

Each of those optionally take a `path` attribute, indicating a subdirectory under the element-defined root to use as the source of files, rather than the root itself. So, a `<files-path>` with a `path="stuff"` attribute would serve files from the `stuff/` subdirectory within `getFilesDir()`. Note that `path` can point to a file as well, to limit access to a single file rather than a directory. Note that `path` is required for `<files-path>`, so you do not accidentally serve everything under `getFilesDir()`.

Also, each can optionally take a `readOnly` attribute. If this is set to `true`, then the files will be readable, but not writeable.

`<external-files-path>` also can take an optional `dir` attribute. If missing, the files are served from `getExternalFilesDir()`. If a valid value of `dir` is supplied, that value is passed into `getExternalFilesDir()`. As such, `dir` is limited to be one of the `Environment.DIRECTORY_*` constants:

- `Alarms`
- `DCIM`
- `Documents`
- `Download`
- `Movies`
- `Music`
- `Notifications`

- Pictures
- Podcasts
- Ringtones

However, you cannot have *both* <external-files-path> with no dir (indicating that you are serving from getExternalFilesDir(null)) *and* one or more <external-files-path> elements with dir values, as they will conflict.

StreamProvider adds support for:

- <raw-resource> for serving a particular raw resource, where the path is the name of the raw resource (without file extension)
- <asset> for serving files from assets/
- <dir-path>, for serving files from locations identified by getDir()
- <external-public-path>, for serving files from locations identified by Environment.getExternalStoragePublicDirectory()

Hence, StreamProvider is especially useful when you want to package some content — such as a PDF file for online help — that you want to serve from your app. Just drop the file in assets/ in your project, set up StreamProvider to serve up assets, and use an appropriate Intent with startActivity() to view that file.

In the case of <dir-path>, two attributes are required:

- dir, which indicates what directory to serve (this is passed into getDir())
- path, which serves its normal role, to determine what to serve from the directory identified by dir

In the case of <external-public-path>, dir is required. It needs to be the string value of one of the Environment.DIRECTORY_* constants, listed above.

Assets and Gradle

For files you are looking to share from your app's assets/, you will need to teach the build system to avoid compressing those files. While annoying, it helps StreamProvider be more compatible with various client apps.

To do this, add an aaptOptions closure to your android closure in your module's build.gradle file. For example, you might have:

```
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.0"

    aaptOptions {
        noCompress 'pdf', 'mp4', 'ogg'
    }
}
```

This would tell Gradle and the build system to not compress files ending in `pdf`, `mp4`, and `ogg`. For your own project, you would choose the file extensions of relevance for the content that you are looking to serve out of `assets/`.

I Can Haz Uri?

`FileProvider` has the `static` `getUriForFile()` convenience method, to build a `Uri` pointing to the `FileProvider`, given the `File` that you wish to serve.

`StreamProvider` has a similar `getUriForFile()` method, with three key differences:

1. It only takes the authority string and the `File`; no `Context` is necessary
2. It only works for files, not assets or raw resources
3. Rather than throwing an exception for an unrecognized `File` (the way `FileProvider` does), `StreamProvider` just returns `null`, indicating that the `File` you requested is not one that the `StreamProvider` is configured to serve

So, you can call `StreamProvider.getUriForFile(AUTHORITY, f)`, for some `String` for your `AUTHORITY` and some `File` (here named `f`) to get a `Uri` pointing to that file, for the purposes of using that `Uri` in an `Intent`, etc.

Uri Prefixes

Activities that support `ACTION_SEND` through an appropriate `<intent-filter>` are likely to have a flaw: they probably do not validate the `Uri` being supplied via `EXTRA_STREAM`. The "surreptitious sharing" attack takes advantage of this, tricking the app into sharing *its own content*. While the researchers who reported this flaw focused on `file:` `Uri` schemes, `content:` is also vulnerable, if your provider's `Uri` values are predictable.

To help defeat this attack, `StreamProvider` automatically adds a per-install UUID to each `Uri`. So, instead of:

```
content://your.authority.here/something/and/a/relative/path.xml
```

the `Uri` will be something like:

```
content://your.authority.here/9b80af30-4507-4f34-956a-3b47e4a7f27f/
something/and/a/relative/path.xml
```

By using a UUID unique for this installation of your app, it makes your `Uri` values dependent upon the device. This makes it more difficult for attackers to hand you a valid `Uri` to your own content to send somewhere that you might not want.

On the flip side, this makes constructing your own `Uri` values a bit more difficult. For files, you can use the `getUriForFile()` method. For assets and raw resources, you can call the `static` `getUriPrefix()` method to get the prefix that is being used, and add that to your `Uri`, such as by using a `Uri.Builder`:

```
PROVIDER
  .buildUpon()
  .appendPath(StreamProvider.getUriPrefix(AUTHORITY))
  .appendPath(path)
  .build()
```

`getUriPrefix()` takes the authority string of your `StreamProvider` and returns the prefix... or `null`, if by subclassing `StreamProvider`, you disabled this prefix.

Extending StreamProvider

You are welcome to create subclasses of `StreamProvider`, to extend its capabilities for things that you may want to do in your app. For example, the instrumentation tests for `StreamProvider` demonstrate creating a subclass that supports serving database files, via a custom `<database-path>` element in the metadata.

By and large, you just create a subclass of `StreamProvider` and use it in your `<provider>` element. Of importance are the hooks in `StreamProvider` to allow subclasses to change critical behavior.

Customizing the Uri Prefix

In your subclass, you have three options for changing the `Uri` prefix used by `StreamProvider`:

- If you want a per-install value, but just not a UUID, override `buildUriPrefix()` and return your own generated `String`
- If you want a fixed prefix, to be used for all installs of this provider, override `getUriPrefix()` and return your constant
- If you do not want a prefix, override `getUriPrefix()` and return `null`

Supporting Other Stream Locations

You may have content located in directories other than what `StreamProvider` supports out of the box, such as the path for SQLite databases. To handle that, you can add support for new XML elements in the `<paths>` element (e.g., `<database-path>` for serving up databases).

To do this, in your `StreamProvider` subclass, override `buildStrategy()` and return a `StreamStrategy` implementation that is configured for your scenario. For files located in unusual spots, `LocalPathStrategy` should work.

In the library's `androidTest/` source set, you will find a `DatabaseProvider` that, at the time of this writing, looks like this:

```
public class DatabaseProvider extends StreamProvider {
  private static final String TAG="database-path";

  @Override
  protected StreamStrategy buildStrategy(Context context,
                                         String tag, String name,
                                         String path, boolean readOnly,
                                         HashMap<String, String> attrs)
    throws IOException {
    if (TAG.equals(tag)) {
      return(new LocalPathStrategy(name,
        context.getDatabasePath(path)));
    }

    return(super.buildStrategy(context, tag, name, path, attrs));
  }
}
```

The parameters to `buildStrategy()` are:

- a `Context`, should you need one (though do not assume it is any particular sort of `Context`)
- the tag we encountered (e.g., `database-path`)
- the value of the `name` attribute, which all of these need to have, as that is how we determine which `StreamStrategy` handles this request
- the value of the `path` attribute, which can be `null`
- a `HashMap` of all attributes, in case you wish to have some custom ones

Either return your own `StreamStrategy` instance based off of this information or chain to the superclass' implementation, so `StreamProvider` can handle the stock tags.

If your provider is intrinsically read-only (i.e., it is impossible to modify the content), you can ignore the `readOnly` flag. If, however, your content could be modified, and you support modification, please honor the `readOnly` flag and block modifications/deletions when that is set to `true`.

Supporting Other Stream Strategies

You may have content located in things that are not files, such as `BLOB` columns in a database. In theory, you can create a custom `StreamStrategy` implementation that handles this. However, this has not been tried much, and so there are likely to be some gaps in the implementation.

That being said, you can examine the built-in strategies (e.g., `AssetStrategy`, `LocalPathStrategy`) and their superclasses (e.g., `AbstractPipeStrategy`) to see how to implement strategies.

Adding Columns to `query()`

You may wish to add other columns in response to a `query()` call, beyond the `OpenableColumns` that `StreamProvider` handles itself and the `_DATA` and `MIME_TYPE` columns added by `LegacyCompatCursorWrapper`.

To do that, override `getValueForQueryColumn()` in your `StreamProvider` subclass. This is supplied the `Uri` of the content and the name of the column requested by the client. You can return an `Object` suitable for stuffing into a `MatrixCursor` to send back – typically, this will be a `String`, `int`, or `long`.

Totally Overhauling Uri Handling

`StreamProvider` itself holds onto a `CompositeStreamStrategy`, delegating all operations to it. If you wish to extend `CompositeStreamStrategy` and do things differently, also override `buildCompositeStrategy()` on your `StreamProvider` subclass, to return the instance of the `CompositeStreamStrategy` that you want the `StreamProvider` to use.

Overriding Standard Methods

You can override standard `ContentProvider` methods (e.g., `getType()`) if needed.

Alternatively, you can override the methods on a `StreamStrategy`, then use that alternative `StreamStrategy` implementation in your `buildStrategy()` method.

Adding Support for insert() and update()

By default, none of the `StreamStrategy` implementations support `insert()` or `update()`. However, your custom `StreamStrategy` can, whether you are extending one of the stock strategy classes or are implementing your own from scratch.

First, override `canInsert()` and/or `canUpdate()`, returning `true` for those operations you do support. Then, you can override `insert()` and `update()`, which have the same method signatures on `StreamStrategy` as they do on `ContentProvider`. There, you can do what you wish.

The Loader Framework

One perpetual problem in Android development is getting work to run outside the main application thread. Every millisecond we spend on the main application thread is a millisecond that our UI is frozen and unresponsive. Disk I/O, in particular, is a common source of such slowdowns, particularly since this is one place where the emulator typically out-performs actual devices. While disk operations rarely get to the level of causing an "application not responding" (ANR) dialog to appear, they can make a UI "janky".

Android 3.0 introduced a new framework to help deal with loading bulk data off of disk, called "loaders". The hope is that developers can use loaders to move database queries and similar operations into the background and off the main application thread. That being said, loaders themselves have issues, not the least of which is the fact that it is new to Android 3.0 and therefore presents some surmountable challenges for use in older Android devices.

This chapter will outline the programming pattern loaders are designed to solve, how to use loaders (both built-in and third-party ones) in your activities, and how to create your own loaders for scenarios not already covered.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- database access
- content provider theory
- content provider implementations

Cursors: Issues with Management

Android had the concept of "managed cursors" in Android 1.x/2.x. A managed `Cursor` was one that an `Activity`... well... manages. More specifically:

1. When the activity was stopped, the managed `Cursor` was deactivated, freeing up all of the memory associated with the result set, and thereby reducing the activity's heap footprint while it was not in the foreground
2. When the activity was restarted, the managed `Cursor` was requeried, to bring back the deactivated data, along the way incorporating any changes in that data that may have occurred while the activity was off-screen
3. When the activity was destroyed, the managed `Cursor` was closed.

This is a delightful set of functionality. `Cursor` objects obtained from a `ContentProvider` via `managedQuery()` were automatically managed; a `Cursor` from `SQLiteDatabase` could be managed by `startManagingCursor()`.

The problem is that the `requery()` operation that was performed when the activity is restarted is executed on the main application thread. As has been noted elsewhere in the book, you *really* do not want to do disk I/O on the main application thread, as it freezes the UI and causes jank. This is particularly true for database I/O, where you may not know in advance exactly how much data you will get back or how long the query will take.

Introducing the Loader Framework

The `Loader` framework was designed to solve three issues with the old managed `Cursor` implementation:

- Arranging for a `requery()` (or the equivalent) to be performed on a background thread)
- Arranging for the original query that populated the data in the first place to also be performed on a background thread, which the managed `Cursor` solution did not address at all
- Supporting loading things other than a `Cursor`, in case you have data from other sources (e.g., XML files, JSON files, Web service calls) that might be able to take advantage of the same capabilities as you can get from a `Cursor` via the loaders

There are three major pieces to the `Loader` framework: `LoaderManager`, `LoaderCallbacks`, and the `Loader` itself.

LoaderManager

`LoaderManager` is your gateway to the `Loader` framework. You obtain one by calling `getLoaderManager()` (or `getSupportLoaderManager()`, as is described later in this chapter). Via the `LoaderManager` you can initialize a `Loader`, restart that `Loader` (e.g., if you have a different query to use for loading the data), etc.

LoaderCallbacks

Much of your interaction with the `Loader`, though, comes from your `LoaderCallbacks` object, such as your activity if that is where you elect to implement the `LoaderCallbacks` interface. Here, you will implement three "lifecycle" methods for consuming a `Loader`:

1. `onCreateLoader()` is called when your activity requests that a `LoaderManager` initialize a `Loader`. Here, you will create the instance of the Loader itself, teaching it whatever it needs to know to go load your data
2. `onLoadFinished()` is called when the `Loader` has actually loaded the data — you can take those results and pour them into your UI, such as calling `swapCursor()` on a `CursorAdapter` to supply the fresh `Cursor`'s worth of data
3. `onLoaderReset()` is called when you should stop using the data supplied to you in the last `onLoadFinished()` call (e.g., the `Cursor` is going to be closed), so you can arrange to make that happen (e.g., call `swapCursor(null)` on a `CursorAdapter`)

When you implement the `LoaderCallbacks` interface, you will need to provide the data type of whatever it is that your `Loader` is loading (e.g., `LoaderCallbacks<Cursor>`). If you have several loaders returning different data types, you may wish to consider implementing `LoaderCallbacks` on multiple objects (e.g., instances of anonymous inner classes), so you can take advantage of the type safety offered by Java generics, rather than implementing `LoaderCallbacks<Object>` or something to that effect.

Loader

Then, of course, there is `Loader` itself.

Consumers of the Loader framework will use some concrete implementation of the abstract `Loader` class in their `LoaderCallbacks` `onCreateLoader()` method. API Level 11 introduced only one concrete implementation: `CursorLoader`, designed to perform queries on a `ContentProvider`, and described in a later section.

Choosing an Implementation

`Loader` and its related classes were introduced in API Level 11 (Android 3.0). If your `minSdkVersion` is 11 or higher, you can use loaders "naturally" via the standard implementation.

If your `minSdkVersion` is below 11, the Android Support package offers its own implementation of `Loader` and the other classes. However, to use it, you will need to work within four constraints:

- You will need to add `support-v4` from the Android Support package as a dependency, directly or indirectly. For example, if you are using `appcompat-v7`, you are already pulling in `support-v4`.
- You will need to inherit from `FragmentActivity`, not the OS base `Activity` class or other refinements (e.g., `MapActivity`), or from other classes that inherit from `FragmentActivity` (e.g., `AppCompatActivity`).
- You will need to import the `support.v4` versions of various classes (e.g., `android.support.v4.app.LoaderManager` instead of `android.app.LoaderManager`)
- You will need to get your `LoaderManager` by calling `getSupportLoaderManager()`, instead of `getLoaderManager()`, on your `FragmentActivity`

These limitations are the same ones that you will encounter when using fragments on older devices. Hence, while loaders and fragments are not really related, you may find yourself adopting both of them at the same time, as part of incorporating the Android Support package into your project.

Using CursorLoader

Let's start off by examining the simplest case: using a `CursorLoader` to asynchronously populate and update a `Cursor` retrieved from a `ContentProvider`. This is illustrated in the `Loaders/ConstantsLoader` sample project, which is the same show-the-list-of-gravity-constants sample application that we examined previously, updated to use the `Loader` framework. Note that this project does not

use the Android Support package and therefore only supports API Level 11 and higher.

In `onCreate()`, rather than executing a `managedQuery()` to retrieve our constants, we ask our `LoaderManager` to initialize a loader, after setting up our `SimpleCursorAdapter` on a null `Cursor`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  adapter=new SimpleCursorAdapter(this,
                        R.layout.row, null,
                        new String[] {Provider.Constants.TITLE,
                                      Provider.Constants.VALUE},
                        new int[] {R.id.title, R.id.value});

  setListAdapter(adapter);
  registerForContextMenu(getListView());
  getLoaderManager().initLoader(0, null, this);
}
```

(from Loaders/ConstantsLoader/app/src/main/java/com/commonsware/android/loader/ConstantsBrowser.java)

Using a `null` `Cursor` means we will have an empty list at the outset, a problem we will rectify shortly.

The `initLoader()` call on `LoaderManager` (retrieved via `getLoaderManager()`) takes three parameters:

- A locally-unique identifier for this loader
- An optional `Bundle` of data to supply to the loader
- A `LoaderCallbacks` implementation to use for the results from this loader (here set to be the activity itself, as it implements the `LoaderManager.LoaderCallbacks<Cursor>` interface)

The first time you call this for a given identifier, your `onCreateLoader()` method of the `LoaderCallbacks` will be called. Here, you need to initialize the `Loader` to use for this identifier. You are passed the identifier plus the `Bundle` (if any was supplied). In our case, we want to use a `CursorLoader`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(this, Provider.Constants.CONTENT_URI,
```

```
                              PROJECTION, null, null, null));
  }
```

(from Loaders/ConstantsLoader/app/src/main/java/com/commonsware/android/loader/ConstantsBrowser.java)

`CursorLoader` takes a `Context` plus all of the parameters you would ordinarily use with `managedQuery()`, such as the content provider `Uri`. Hence, converting existing code to use `CursorLoader` means converting your `managedQuery()` call into an invocation of the `CursorLoader` constructor inside of your `onCreateLoader()` method.

At this point, the `CursorLoader` will query the content provider, but do so on a background thread, so the main application thread is not tied up. When the `Cursor` has been retrieved, it is supplied to your `onLoadFinished()` method of your `LoaderCallbacks`:

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
  adapter.swapCursor(cursor);
}
```

(from Loaders/ConstantsLoader/app/src/main/java/com/commonsware/android/loader/ConstantsBrowser.java)

Here, we call the new `swapCursor()` available on `CursorAdapter`, to replace the original `null` `Cursor` with the newly-loaded `Cursor`.

Your `onLoadFinished()` method will also be called whenever the data represented by your `Uri` changes. That is because the `CursorLoader` is registering a `ContentObserver`, so it will find out about data changes and will automatically requery the `Cursor` and supply you with the updated data.

Eventually, `onLoaderReset()` will be called. You are passed a `Cursor` object that you were supplied previously in `onLoadFinished()`. You need to make sure that you are no longer using that `Cursor` at this point — in our case, we swap `null` back into our `CursorAdapter`:

```
public void onLoaderReset(Loader<Cursor> loader) {
  adapter.swapCursor(null);
}
```

(from Loaders/ConstantsLoader/app/src/main/java/com/commonsware/android/loader/ConstantsBrowser.java)

And that's pretty much it, at least for using `CursorLoader`. Of course, you need a content provider to make this work, and creating a content provider involves a bit of work.

What Else Is Missing?

The `Loader` framework does an excellent job of handling queries in the background. What it does not do is help us with anything else that is supposed to be in the background, such as inserts, updates, deletes, or creating/upgrading the database. It is all too easy to put those on the main application thread and therefore possibly encounter issues. Moreover, since the thread(s) used by the `Loader` framework are an implementation detail, we cannot use those threads ourselves necessarily for the other CRUD operations.

What Happens When…?

Here are some common development scenarios and how the `Loader` framework addresses them.

… the Data Behind the Loader Changes?

According to the `Loader` documentation, "They monitor the source of their data and deliver new results when the content changes".

The documentation is incorrect.

A `Loader` *can* "monitor the source of their data and deliver new results when the content changes". There is nothing in the framework that requires this behavior. Moreover, there are some cases where it is clearly a bad idea to do this — imagine a `Loader` loading data off of the Internet, needing to constantly poll some server to look for changes.

The documentation for a Loader implementation should tell you the rules. Android's built-in `CursorLoader` does deliver new results, by means of a behind-the-scenes `ContentObserver`. However, it is not automatic that a `Loader` deliver new results, and it may be impractical for a `Loader` to deliver new results.

… the Configuration Changes?

The managed `Cursor` system that the `Loader` framework replaces would automatically `requery()` any managed `Cursor` objects when an activity was restarted. This would update the `Cursor` in place with fresh data after a

configuration change. Of course, it would do that on the main application thread, which was not ideal.

Your Loader objects are retained across the configuration change automatically. Barring bugs in a specific `Loader` implementation, your Loader should then hand the new activity instance the data that was retrieved on behalf of the old activity instance (e.g., the `Cursor`).

Hence, you do not have to do anything special for configuration changes.

… the Activity is Destroyed?

Another thing the managed `Cursor` system gave you was the automatic closing of your `Cursor` when the activity was destroyed. The `Loader` framework does this as well, by triggering a reset of the `Loader`, which obligates the `Loader` to release any loaded data.

… the Activity is Stopped?

The final major feature of the managed `Cursor` system was that it would `deactivate()` a managed `Cursor` when the activity was stopped. This would release all of the heap space held by that `Cursor` while it was not on the screen. Since the `Cursor` was refreshed as part of restarting the activity, this usually worked fairly well and would help minimize pressure on the heap.

Alas, this does not appear to be supported by the `Loader` framework. The `Loader` is reset when an activity is destroyed, not stopped. Hence, the `Loader` data will continue to tie up heap space even while the activity is not in the foreground.

For many activities, this should not pose a problem, as the heap space consumed by their `Cursor` objects is modest. If you have an activity with a massive `Cursor`, though, you may wish to consider what steps you can take on your own, outside of the `Loader` framework, to help with this.

Writing a Custom Loader

Perhaps, despite the above issues, and despite the author's assertion that the `Loader` framework is a failed abstraction, you want to implement a custom `Loader`.

You have two main choices for doing that:

- If the API that you are using is intrinsically asynchronous, you can extend `Loader`
- If the API that you are using is synchronous, most likely you should extend `AsyncTaskLoader`, which manages an `AsyncTask` for you, giving you a background thread for loading the content

If your API can work either way — synchronously or asynchronously – either option works. In terms of getting the `Loader` implementation right, you may want to use `AsyncTaskLoader`, as it will ensure that everything is delivered on the right thread. On the other hand, you may have greater control over the nature of the asynchronous work using the API's native asynchronous capability, such as configuring a thread pool.

The `HTTP/RetroLoader` sample project is a clone of the `HTTP/Retrofit` sample app from the chapter on Internet access. However, this time, the Retrofit work to load the most recent `android` Stack Overflow questions will be mediated by a `QuestionsLoader`.

Changing the Retrofit Interface

Retrofit offers both synchronous and asynchronous APIs. For the purposes of this sample, we will use the synchronous API, to see how one might implement an `AsyncTaskLoader`. That, in turn, requires us to modify `StackOverflowInterface`, having `questions()` return the `SOQuestions` directly, rather than by using a callback:

```
package com.commonsware.android.retrofit;

import retrofit.http.GET;
import retrofit.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  SOQuestions questions(@Query("tagged") String tags);
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/StackOverflowInterface.java)

Otherwise, this is unchanged from the original edition of this sample.

Implementing QuestionsLoader

`QuestionsLoader` is an implementation of `AsyncTaskLoader`. The documentation for `Loader` and `AsyncTaskLoader` leave a lot to be desired. `QuestionsLoader` is based on "triangulation" between the installed-applications loader included in the `AsyncTaskLoader` documentation and the source code to `CursorLoader`.

`Loader` uses Java generics. Its declaration requires the type of content being loaded by the `Loader`. So, `QuestionsFragment` is a `Loader` of `SOQuestions`:

```
public class QuestionsLoader extends AsyncTaskLoader<SOQuestions> {
  final private StackOverflowInterface so;
  private SOQuestions lastResult;

  public QuestionsLoader(Context context) {
    super(context);

    RestAdapter restAdapter=
      new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com")
        .build();

    so=restAdapter.create(StackOverflowInterface.class);
  }

  @Override
  protected void onStartLoading() {
    super.onStartLoading();

    if (lastResult!=null) {
      deliverResult(lastResult);
    }
    else {
      forceLoad();
    }
  }

  @Override
  synchronized public SOQuestions loadInBackground() {
    if (isLoadInBackgroundCanceled()) {
      throw new OperationCanceledException();
    }

    return(so.questions("android"));
  }

  @Override
```

```
  public void deliverResult(SOQuestions data) {
    if (isReset()) {
      // actual cleanup, if any
    }

    lastResult=data;

    if (isStarted()) {
      super.deliverResult(data);
    }
  }

  @Override
  protected void onStopLoading() {
    super.onStopLoading();

    cancelLoad();
  }

  @Override
  protected void onReset() {
    super.onReset();

    onStopLoading();
    // plus any actual cleanup
  }
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

The Constructor

A `Loader` needs to have a constructor that takes a `Context` as a parameter. So, `QuestionsLoader` has one, that chains to the superclass constructor, plus sets up the `StackOverflowInterface` using Retrofit:

```
  public QuestionsLoader(Context context) {
    super(context);

    RestAdapter restAdapter=
      new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com")
        .build();

    so=restAdapter.create(StackOverflowInterface.class);
  }
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

Loading the Questions

The real work for loading the questions comes in `loadInBackground()`. Subclasses of `AsyncTaskLoader` need to implement this to return the content being loaded. As the name suggests, `loadInBackground()` is called on a background thread, so you can take time here.

The key piece of the `loadInBackground()` of `QuestionsFragment` is the call to `questions()` on the `StackOverflowInterface`, to retrieve the desired questions:

```
@Override
synchronized public SOQuestions loadInBackground() {
  if (isLoadInBackgroundCanceled()) {
    throw new OperationCanceledException();
  }

  return(so.questions("android"));
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

However, it is possible that before `loadInBackground()` is called, that something cancels the `AsyncTask` in the `AsyncTaskLoader`. A subclass of `AsyncTaskLoader` needs to check `isLoadInBackgroundCanceled()` in `loadInBackground()`, and throw an `OperationCanceledException` if `isLoadInBackgroundCanceled()` returns `true`.

Delivering Results

Not only do subclasses of `AsyncTaskLoader` have to load the content, they also have to cache the results of the previous load. So, `QuestionsLoader` has a `lastResult` field, holding onto an `SOQuestions` object.

In `deliverResult()`, we need to do three things:

1. If `isReset()` returns `true`, indicating that the `Loader` was reset, we need to clean up anything associated with the previous load results
2. We need to cache the new load results, which are passed into `deliverResult()` as a parameter
3. If the loader is started (`isStarted()` returns `true`), we need to chain to the superclass implementation of `deliverResult()` to actually deliver the data to the `LoaderCallbacks` implementation:

```
@Override
public void deliverResult(SOQuestions data) {
  if (isReset()) {
    // actual cleanup, if any
  }

  lastResult=data;

  if (isStarted()) {
    super.deliverResult(data);
  }
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

Here, `lastResult` is just a POJO holding onto other POJOs, so there is nothing specific for us to do in case the loader was reset. If our results were a `Cursor`, a `Bitmap`, or other objects with clear "close" or "release" semantics, you might do that work if `onReset()` returned `true`.

Starting, Stopping, and Resetting

You also need to implement three additional methods.

First is `onStartLoading()`. Here is where we use the cached result, delivering it via `deliverResults()`. If we do not have a cached result, we need to call `forceLoad()` to trigger the `AsyncTask` which, in turn, triggers `loadInBackground()` and the rest of the work to actually retrieve the results:

```
@Override
protected void onStartLoading() {
  super.onStartLoading();

  if (lastResult!=null) {
    deliverResult(lastResult);
  }
  else {
    forceLoad();
  }
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

There is a corresponding `onStopLoading()`, where we need to call `cancelLoad()`, to cancel the `AsyncTask`:

```
  @Override
  protected void onStopLoading() {
    super.onStopLoading();

    cancelLoad();
  }
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

Finally, there is `onReset()`, where we need to call `onStopLoading()` (as loading should stop if the `Loader` is reset), plus do any cleanup of our cached results as needed:

```
  @Override
  protected void onReset() {
    super.onReset();

    onStopLoading();
    // plus any actual cleanup
  }
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsLoader.java)

Using QuestionsLoader

You might think that with 70-odd lines of `QuestionsLoader` code that we would have a corresponding savings in `QuestionsFragment`.

Alas, no, though it is a bit shorter.

`QuestionsFragment` now implements `LoaderCallbacks` instead of Retrofit's `Callback` interface:

```
public class QuestionsFragment extends ListFragment implements
    LoaderManager.LoaderCallbacks<SOQuestions> {
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

In the original `QuestionsFragment`, we fired off the asynchronous Retrofit work in `onCreateView()`, and we processed the results in the `success()` and `failure()` methods.

Now, we just need to call `initLoader()`, in this case from `onViewCreated()`:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  getLoaderManager().initLoader(0, null, this);
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

Then, our `LoaderCallbacks` methods create the `QuestionsLoader` and apply the results:

```
@Override
public Loader<SOQuestions> onCreateLoader(int id, Bundle args) {
  return(new QuestionsLoader(getActivity()));
}

@Override
public void onLoadFinished(Loader<SOQuestions> loader,
                           SOQuestions data) {
  setListAdapter(new ItemsAdapter(data.items));
}

@Override
public void onLoaderReset(Loader<SOQuestions> loader) {
  setListAdapter(null);
}
```

(from HTTP/RetroLoader/app/src/main/java/com/commonsware/android/retrofit/QuestionsFragment.java)

Considering the Loader Contract

As noted earlier in this chapter, there are three key pieces of the `Loader` contract: asynchronicity, data retention, and automatic delivery of updates on content changes.

So, how does `QuestionsLoader` stack up?

- `AsyncTaskLoader` provides the asynchronous operation for us, so we inherit that
- All loaders are automatically retained by the `LoaderManager`, so there is nothing specific that we need to do for that
- However, we are unaware of any changes in our content, such as new questions being asked, so we are not delivering updated content to clients of the `QuestionsLoader`

This is a common gap with loader implementations. Occasionally, we may be in position to find out when the content changes. More often, we are not, or the work to find out about content changes has to be handled by a much larger subsystem, beyond a simple `Loader` subclass.

For example, if we *really* wanted to have `QuestionsLoader` automatically deliver a fresh `SOQuestions` object when new Stack Overflow `android` questions were asked, we could: use some sort of timing mechanism (e.g., `ScheduledExecutorService`) to poll the Stack Exchange API every so often. However, then we would need to implement some sort of "diff" algorithm to determine if relevant data changed in the JSON response, so we knew to deliver a fresh `SOQuestions` to clients. However, our polling period would be somewhat arbitrary. Frequent polling would consume a fair amount of battery and bandwidth as well.

The ContactsContract and CallLog Providers

One of the more popular stores of data on your average Android device is the contact list. Ever since Android 2.0, Android tracks contacts across multiple different "accounts", or sources of contacts. Some may come from your Google account, while others might come from Exchange or other services.

This chapter will walk you through some of the basics for accessing the contacts on the device. Along the way, we will revisit and expand upon our knowledge of using a `ContentProvider`.

First, we will review the contacts APIs, past and present. We will then demonstrate how you can connect to the contacts engine to let users pick and view contacts... all without your application needing to know much of how contacts work. We will then show how you can query the contacts provider to obtain contacts and some of their details, like email addresses and phone numbers. We wrap by showing how you can invoke a built-in activity to let the user add a new contact, possibly including some data supplied by your application.

In addition, we will take a peek at the `CallLog` provider, which, as the name suggests, gives you access to a log of calls made on the device.

Prerequisites

Understanding this chapter requires that you have read these chapters in addition to the core chapters:

- content provider theory

- content provider implementations
- the Loader framework

Introducing You to Your Contacts

Android makes contacts available to you via a complex ContentProvider framework, so you can access many facets of a contact's data — not just their name, but addresses, phone numbers, groups, etc. Working with the contacts ContentProvider set is simple... only if you have an established pattern to work with. Otherwise, it may prove somewhat daunting.

Organizational Structure

The contacts ContentProvider framework can be found as the set of ContactsContract classes and interfaces in the android.provider package. Unfortunately, there is a dizzying array of inner classes to ContactsContract.

Contacts can be broken down into two types: raw and aggregate. Raw contacts come from a sync provider or are hand-entered by a user. Aggregate contacts represent the sum of information about an individual culled from various raw contacts. For example, if your Exchange sync provider has a contact with an email address of jdoe@foo.com, and your Facebook sync provider has a contact with an email address of jdoe@foo.com, Android may recognize that those two raw contacts represent the same person and therefore combine those in the aggregate contact for the user. The classes relating to raw contacts usually have Raw somewhere in their name, and these normally would be used only by custom sync providers.

The ContactsContract.Contacts and ContactsContract.Data classes represent the "entry points" for the ContentProvider, allowing you to query and obtain information on a wide range of different pieces of information. What is retrievable from these can be found in the various ContactsContract.CommonDataKinds series of classes. We will see examples of these operations later in this chapter.

A Look Back at Android 1.6

Prior to Android 2.0, Android had no contact synchronization built in. As a result, all contacts were in one large pool, whether they were hand-entered by users or were added via third-party applications. The API used for this is the Contacts ContentProvider.

The `Contacts` `ContentProvider` still works, as it is merely deprecated in Android 2.0.1, not removed. In practice, it has one big limitation: it will only report contacts added directly to the device (as opposed to ones synchronized from Microsoft Exchange, Facebook, or other sources). As a result, modern Android apps should **not** be using `Contacts` in general — use `ContactsContract`.

Pick a Peck of Pickled People

Back in the chapter on resource sets and configurations, we saw a series of examples of handling configuration changes. Those samples allowed the user to pick a contact and view a contact. There, we focused on the configuration change aspect. Here, let's examine the actual pick and view logic a bit more closely.

Picking a Contact

When the user picks a contact, we call `startActivityForResult()` with an `ACTION_PICK` `Intent`:

```
public void pickContact(View v) {
  Intent i=
      new Intent(Intent.ACTION_PICK,
                 ContactsContract.Contacts.CONTENT_URI);

  startActivityForResult(i, PICK_REQUEST);
}
```

(from ConfigChange/Fragments/app/src/main/java/com/commonsware/android/rotation/frag/RotationFragment.java)

The `Intent` has `ContactsContract.Contacts.CONTENT_URI` as its `Uri`. Here, `ContactsContract.Contacts.CONTENT_URI` is defined by the Android SDK and points to the contacts "table" inside the `ContactsContract` "database", as it were. Whether there is really a database or a table involved is up to the implementation of `ContactsContract`, of course.

When we call `startActivityForResult()`, Android needs to find an activity to fulfill this request. However, at the outset, all it has is an action string and a `Uri`. There could be all sorts of activities on the device that advertise that they can pick from a collection identified by a `Uri` starting with the `content` scheme.

To help refine the request, Android asks the `ContactsContract` `ContentProvider` what the MIME type is for this `Uri`. Then, Android knows an action string, a MIME type, and a `Uri`. It so happens that the contacts apps that ship on Android have an

activity that has an `<intent-filter>` that indicates that it can handle `ACTION_PICK` of the relevant MIME type from a `content` `Uri`. And so that is the activity that the user sees.

The `Uri` that we get back in `onActivityResult()` not only points to the contact that the user picked, but also gives us *temporary read access* to that contact's personally identifying information. In effect, it is as if the normal `READ_CONTACTS` permission requirement was suspended, for this one `Uri`, for our app alone. Once our process terminates, we may no longer have the ability to get at details about that contact via its `Uri`, as this read access is temporary.

Viewing a Contact

As it turns out, the sample app does not take advantage of the temporary read access. Instead, when the user clicks the "View" button, the app just brings up an activity to go view that contact:

```
public void viewContact(View v) {
  startActivity(new Intent(Intent.ACTION_VIEW, contact));
}
```

(from ConfigChange/Fragments/app/src/main/java/com/commonsware/android/rotation/frag/RotationFragment.java)

Once again, Android has an action string (`ACTION_VIEW`) and a `Uri` (the one that we got in response to the `ACTION_PICK` request). And, once again, Android asks `ContactsContract` for the MIME type of the data associated with this `Uri`, so that the MIME type can help identify the right activity to handle this request. The contacts app that comes on the device should have an activity that complies, and so we can view the contact.

In truth, the only reason why we as developers can count on these activities existing is because of Google Play Services and the Play Store. The Compatibility Definition Document (CDD) that manufacturers must comply with to get Google's proprietary Android apps requires that the device ship with apps that fulfill all of the `<intent-filter>` elements supported by the apps in the Android Open Source Project (AOSP). Hence, for devices that legitimately have the Play Store on them, there should always be an app that offers activities to allow users to pick and view contacts. However, on devices that do *not* legitimately have the Play Store, those activities might not exist. Manufacturers who avoid Google's proprietary apps *should* still aim to comply with the CDD as much as possible, if they want third-party apps like yours to work successfully on those devices. However, there is no contractual

requirement that they do, and so, as the saying goes, your mileage may vary (YMMV).

Spin Through Your Contacts

The preceding example allows you to work with contacts, yet not actually have any contact data other than a transient `Uri`. All else being equal, it is best to use the contacts system this way, as it means you do not need any extra permissions that might raise privacy issues.

Of course, all else is rarely equal.

Your alternative, therefore, is to execute queries against the contacts `ContentProvider` to get actual contact detail data back, such as names, phone numbers, and email addresses. The `Contacts/Spinners` sample application will demonstrate this technique.

Contact Permissions

Since contacts are privileged data, you need certain permissions to work with them. Specifically, you need the `READ_CONTACTS` permission to query and examine the `ContactsContract` content and `WRITE_CONTACTS` to add, modify, or remove contacts from the system. This only holds true if your code will have access to personally-identifying information, which is why the `Pick` sample above — which just has an opaque `Uri` — does not need any permission.

For example, here is the manifest for the `Contacts/Spinners` sample application:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.contacts.spinners"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name=".ContactSpinners"
```

```
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

(from Contacts/Spinners/app/src/main/AndroidManifest.xml)

And, since this app has a `targetSdkVersion` of 26, we also need to deal with runtime permissions, as `READ_CONTACTS` has a `protectionLevel` of `dangerous`, so we need to ask the user for permission at runtime. To that end, we use the same `AbstractPermissionActivity` seen elsewhere in the book as the base class for our `ContactSpinners` activity, so we can delegate all of the runtime permission logic to `AbstractPermissionActivity`. The one difference is that this sample originally used a `ListActivity`, so this sample's edition of `AbstractPermissionActivity` inherits from `ListActivity` instead of from `Activity`.

Pre-Joined Data

While the database underlying the `ContactsContract` content provider is private, one can imagine that it has several tables: one for people, one for their phone numbers, one for their email addresses, etc. These are tied together by typical database relations, most likely 1:N, so the phone number and email address tables would have a foreign key pointing back to the table containing information about people.

To simplify accessing all of this through the content provider interface, Android pre-joins queries against some of the tables. For example, you can query for phone numbers and get the contact name and other data along with the number — you do not have to do this join operation yourself.

The UI

The `ContactSpinners` activity has the standard `ListView` along with a `Spinner`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    >
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

(from Contacts/Spinners/app/src/main/res/layout/main.xml)

In `onReady()` of the activity, we load up the `Spinner`:

```
  @Override
  public void onReady() {
    setContentView(R.layout.main);

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              getResources().getStringArray(R.array.options));

    aa.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

In particular, we populate the `Spinner` based on a `<string-array>` resource from the `res/values/arrays.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="options">
    <item>Contact Names</item>
    <item>Contact Names &amp;amp; Numbers</item>
    <item>Contact Names &amp;amp; Email Addresses</item>
  </string-array>
</resources>
```

(from Contacts/Spinners/app/src/main/res/values/arrays.xml)

Reacting to the Spinner

We set up the activity to be the `OnItemSelectedListener` for the `Spinner`, which means that we have to implement `onItemSelected()` and `onNothingSelected()`:

```
@Override
public void onItemSelected(AdapterView<?> parent,
                           View v, int position, long id) {
  getLoaderManager().initLoader(position, null, this);
}

@Override
public void onNothingSelected(AdapterView<?> parent) {
  // ignore
}
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

When the user selects something in the `Spinner` — and for the default selection — we will use the `Loader` framework and use a `CursorLoader` to query the `ContactsContract` `ContentProvider`. In this case, though, we want three different `Cursor` values, one for each option in the `Spinner`. That will mean that we need three different `CursorLoader` objects. To identify which loader we are going to initialize, we pass in the `position` of the `Spinner` to `initLoader()`, so the 0/1/2 value that we get as the `position` forms our loader ID.

Loading the Data

In `onCreateLoader()` of our `LoaderCallbacks`, we need to return a `CursorLoader` for whichever `loaderId` was passed in. What varies is the `Uri` that we want to query and the "projection" of "columns" that we want to get back. So, `onCreateLoader()` uses a `switch` statement to decide what `Uri` and projection to use, then creates a `CursorLoader` based upon that:

```
@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  String[] projection;
  Uri uri;

  switch (loaderId) {
    case LOADER_NAMES:
      projection=PROJECTION_NAMES;
```

```
      uri=ContactsContract.Contacts.CONTENT_URI;
      break;

    case LOADER_NAMES_NUMBERS:
      projection=PROJECTION_NUMBERS;
      uri=ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
      break;

    default:
      projection=PROJECTION_EMAILS;
      uri=ContactsContract.CommonDataKinds.Email.CONTENT_URI;
      break;
  }

  return(new CursorLoader(this, uri, projection, null, null,
                          ContactsContract.Contacts.DISPLAY_NAME));
}
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

The two `case` values are just constants tied to the positions from the `Spinner`, defined as static data members:

```
private static final int LOADER_NAMES=0;
private static final int LOADER_NAMES_NUMBERS=1;
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

Similarly, the three projections are defined as static data members:

```
private static final String[] PROJECTION_NAMES=new String[] {
    ContactsContract.Contacts._ID,
    ContactsContract.Contacts.DISPLAY_NAME,
};
private static final String[] PROJECTION_NUMBERS=new String[] {
    ContactsContract.Contacts._ID,
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Phone.NUMBER
};
private static final String[] PROJECTION_EMAILS=new String[] {
    ContactsContract.Contacts._ID,
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Email.DATA
};
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

For the "names" `Spinner` entry, we are going to retrieve the ID and display name of the contact, using the standard `ContactsContract.Contacts.CONTENT_URI` `Uri` value.

For the "names and phone numbers" `Spinner` entry, we still want the display name of the contact, but we also want phone numbers. Fortunately, as mentioned earlier, `ContactsContract` denormalizes its data in response to queries, so we can get the display name of the contact even when we are querying the "table" of phone numbers, via `ContactsContract.CommonDataKinds.Phone.CONTENT_URI`. The same basic process holds true for the "names and emails" entry, where we query `ContactsContract.CommonDataKinds.Email.CONTENT_URI`. Note that we will get somewhat redundant information back — if a contact has two phone numbers, we get two rows in our `Cursor`, both for the same contact, and one per phone number.

We can sort by `DISPLAY_NAME` for all three cases, courtesy of the aforementioned denormalization of the data.

Showing the Results

We also have to implement `onLoadFinished()`, to take in the `Cursor` that is the result of the query against `ContactsContract` and put the results in the `ListView`. Once again, the rendering will differ a bit based upon whether we are showing just names or names along with other data (e.g., phone numbers). So, we have another `switch` statement, where we determine what columns we want, what layout ID to use, and what roster of widgets in that layout map to those columns:

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
  String[] columns;
  int layoutId;
  int[] views;

  switch(loader.getId()) {
    case LOADER_NAMES:
      columns=COLUMNS_NAMES;
      layoutId=android.R.layout.simple_list_item_1;
      views=VIEWS_ONE;
      break;

    case LOADER_NAMES_NUMBERS:
      columns=COLUMNS_NUMBERS;
      layoutId=android.R.layout.simple_list_item_2;
      views=VIEWS_TWO;
```

```
      break;

    default:
      columns=COLUMNS_EMAILS;
      layoutId=android.R.layout.simple_list_item_2;
      views=VIEWS_TWO;
      break;
  }

  setListAdapter(new SimpleCursorAdapter(this, layoutId, c,
                                         columns, views, 0));
}
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

The lists of columns and views are defined as static data members and map positionally (i.e., the first view is for the first column):

```
private static final String[] COLUMNS_NAMES=new String[] {
  ContactsContract.Contacts.DISPLAY_NAME
};
private static final String[] COLUMNS_NUMBERS=new String[] {
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Phone.NUMBER
};
private static final String[] COLUMNS_EMAILS=new String[] {
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Email.DATA
};
private static final int[] VIEWS_ONE=new int[] {
    android.R.id.text1
};
private static final int[] VIEWS_TWO=new int[] {
    android.R.id.text1,
    android.R.id.text2
};
```

(from Contacts/Spinners/app/src/main/java/com/commonsware/android/contacts/spinners/ContactSpinners.java)

Then, we create a `SimpleCursorAdapter` wrapped around that information and use that to populate the `ListView`, thereby showing the contacts and the requested information about those contacts.

Makin' Contacts

Let's now take a peek at the reverse direction: adding contacts to the system. This was never particularly easy and now is... well, different.

First, we need to distinguish between sync providers and other apps. Sync providers are the guts underpinning the accounts system in Android, bridging some existing source of contact data to the Android device. Hence, you can have sync providers for Exchange, Facebook, and so forth. These will need to create raw contacts for newly-added contacts to their backing stores that are being sync'd to the device for the first time. Creating sync providers is outside of the scope of this book for now.

It is possible for other applications to create contacts. These, by definition, will be phone-only contacts, lacking any associated account, no different than if the user added the contact directly. The recommended approach to doing this is to collect the data you want, then spawn an activity to let the user add the contact — this avoids your application needing the `WRITE_CONTACTS` permission and all the privacy/data integrity issues that creates.

To that end, take a look at the `Contacts/Inserter` sample project. It defines a simple activity with a two-field UI, with one field apiece for the person's first name and phone number:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
  >
  <TableRow>
    <TextView
      android:text="First name:"
    />
    <EditText android:id="@+id/name"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Phone:"
    />
    <EditText android:id="@+id/phone"
      android:inputType="phone"
```

```
    />
  </TableRow>
  <Button android:id="@+id/insert" android:text="Insert!" />
</TableLayout>
```

(from Contacts/Inserter/app/src/main/res/layout/main.xml)

The trivial UI also sports a button to add the contact:

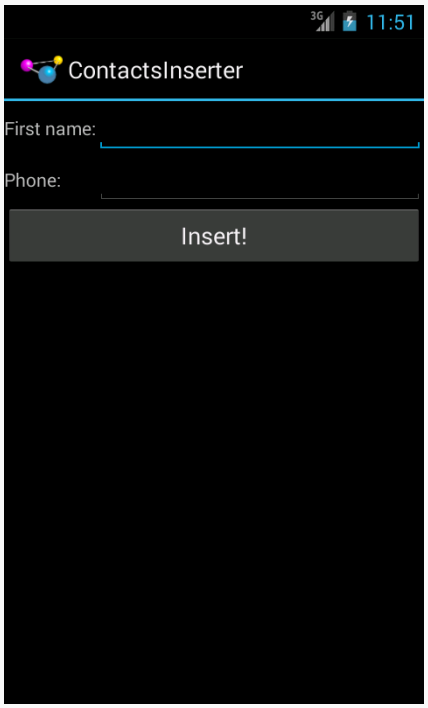


Figure 772: The ContactInserter sample application

When the user clicks the button, the activity gets the data and creates an `Intent` to be used to launch the add-a-contact activity. This uses the `ACTION_INSERT_OR_EDIT` action and a couple of extras from the `ContactsContract.Intents.Insert` class:

```
package com.commonsware.android.inserter;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.Intents.Insert;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
```

```
public class ContactsInserter extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Button btn=(Button)findViewById(R.id.insert);

    btn.setOnClickListener(onInsert);
  }

  View.OnClickListener onInsert=new View.OnClickListener() {
    public void onClick(View v) {
      EditText fld=(EditText)findViewById(R.id.name);
      String name=fld.getText().toString();

      fld=(EditText)findViewById(R.id.phone);

      String phone=fld.getText().toString();
      Intent i=new Intent(Intent.ACTION_INSERT_OR_EDIT);

      i.setType(Contacts.CONTENT_ITEM_TYPE);
      i.putExtra(Insert.NAME, name);
      i.putExtra(Insert.PHONE, phone);
      startActivity(i);
    }
  };
}
```

(from Contacts/Inserter/app/src/main/java/com/commonsware/android/inserter/ContactsInserter.java)

We also need to set the MIME type on the `Intent` via `setType()`, to be `CONTENT_ITEM_TYPE`, so Android knows what sort of data we want to actually insert. Then, we call `startActivity()` on the resulting `Intent`. That brings up an add-or-edit activity:

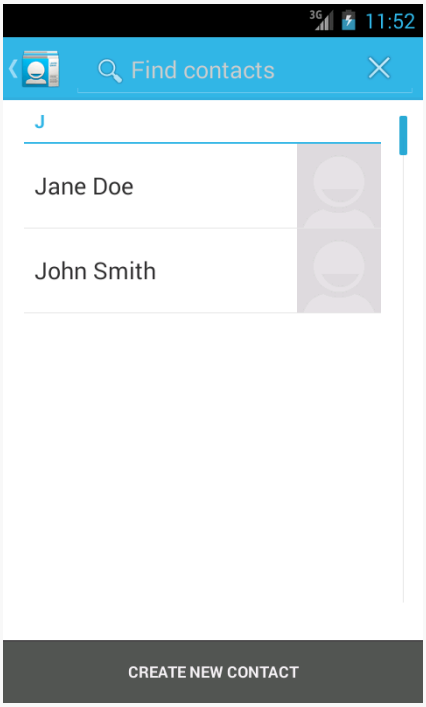


Figure 773: The add-or-edit-a-contact activity

... where if the user chooses "Create new contact", they are taken to the ordinary add-a-contact activity, with our data pre-filled in:

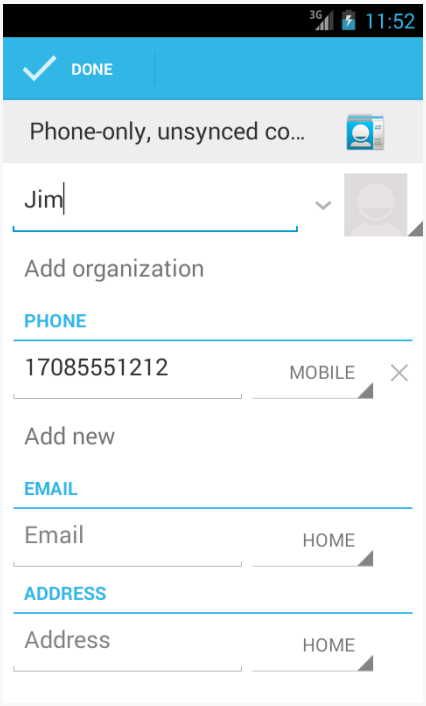


Figure 774: The edit-contact form, showing the data from the ContactInserter activity

Note that the user could choose an existing contact, rather than creating a new contact. If they choose an existing contact, the first name of that contact will be overwritten with the data supplied by the `ContactsInserter` activity, and a new phone number will be added from those `Intent` extras.

Looking at the CallLog

A closely-related `ContentProvider` to `ContactsContract` is `CallLog`. As the name suggests, it contains a log of calls for this device, including things like the date/time of the call, the call duration, and the other party on the call (e.g., a phone number).

If you wish to give the user another look at their calls, independent from the UI available on the device (e.g., Dialer app), you might wish to query the `CallLog`, as we do in the `Contacts/CallLog` sample application

Pondering Permissions

In the beginning, there was no `READ_CALL_LOG` permission. To read the `CallLog` provider, you needed to hold `READ_CONTACTS`. The reason for the `READ_CONTACTS`

permission is that the CallLog denormalizes the data, copying into its own table contact data about the other party, so that the CallLog can remain independent of ContactsContract.

In API Level 16, though, they added the READ_CALL_LOG permission. If your minSdkVersion is 16 or higher, you can just request READ_CALL_LOG. If your minSdkVersion is lower than that, though, you will want to request both permissions, to make sure that you are covered. You might consider using the android:maxSdkVersion attribute on the READ_CONTACTS <uses-permission> element, as you will not need it on newer devices, unless you are also working with ContactContract.

Both READ_CALL_LOG and READ_CONTACTS are dangerous permissions, and therefore if your targetSdkVersion is 23 or higher, you need to request those permissions at runtime. Of course, you only need to request the permissions that you need, and so if your minSdkVersion is 16 or higher and you are only using READ_CALL_LOG, you only need to request READ_CALL_LOG at runtime.

Our sample app uses the same AbstractPermissionActivity as did the first sample app in this chapter, though this time our CallLogConsumerActivity will request READ_CALL_LOG instead of READ_CONTACTS.

Contents of CallLog.Calls

The sample app requests the READ_CALL_LOG permission, so it can query the CallLog:

```
  <uses-permission android:name="android.permission.READ_CALL_LOG" />
```

(from Contacts/CallLog/app/src/main/AndroidManifest.xml)

The app has one Java class, CallLogConsumerActivity, which is the launcher activity.

In onCreate() — among other bits of work that we will explore shortly – we call getLoaderManager().initLoader(), to query the CallLog via a CursorLoader. The activity itself implements the LoaderManager.LoaderCallbacks interface needed by initLoader(), and so the activity has the three required LoaderCallbacks methods:

```
  @Override
  public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    return(new CursorLoader(this, CallLog.Calls.CONTENT_URI,
```

```
                            PROJECTION, null, null, CallLog.Calls.DATE
                                + " DESC"));
  }

  @Override
  public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
  }

  @Override
  public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
  }
```

(from Contacts/CallLog/app/src/main/java/com/commonsware/android/calllog/consumer/CallLogConsumerActivity.java)

Here, we retrieve the data from the `CallLog.Calls` "table" via its `CONTENT_URI`, asking for the "columns" indicated by the `PROJECTION`:

```
private static final String[] PROJECTION=new String[] {
    CallLog.Calls._ID, CallLog.Calls.NUMBER, CallLog.Calls.DATE };
```

(from Contacts/CallLog/app/src/main/java/com/commonsware/android/calllog/consumer/CallLogConsumerActivity.java)

We sort the data descending by date. It would be nice if the documentation for `CallLog` included some indication that this approach was endorsed and supported. Based on the `CallLog` implementation, it should be stable.

The `Cursor` itself is passed into a `SimpleCursorAdapter` (named `adapter`) via `swapCursor()` calls.

Showing the CallLog

In `onReady()`, we want to set up that `SimpleCursorAdapter` for mapping the phone number and date of a call to corresponding `TextView` widgets in a row layout:

```
@Override
public void onReady() {
  adapter=
      new SimpleCursorAdapter(this, R.layout.row, null, new String[] {
          CallLog.Calls.NUMBER, CallLog.Calls.DATE }, new int[] {
          R.id.number, R.id.date }, 0);

  adapter.setViewBinder(this);
  setListAdapter(adapter);
```

```
    getLoaderManager().initLoader(0, null, this);
  }
```

(from Contacts/CallLog/app/src/main/java/com/commonsware/android/calllog/consumer/CallLogConsumerActivity.java)

Here, we create the `SimpleCursorAdapter` on a `null` `Cursor` at the outset, to indicate that we do not yet have our data, and we will show the `Cursor` delivered to `onLoadFinished()` at that time.

However we will run into a problem with the date. In the `CallLog` provider, the date is stored as "milliseconds since the Unix epoch", the same time system used by `System.currentTimeMillis()`. That is a really long number, one that ordinary people will not recognize. If we blindly just convert that into a string and put it in the `TextView`, users will be unable to understand that column.

To get a chance to convert that value into something more useful, the activity implements the `SimpleCursorAdapter.ViewBinder` interface and calls `setViewBinder(this)` on the `adapter`. A `ViewBinder` will get control every time the `SimpleCursorAdapter` tries binding a value from the `Cursor` to a widget, so we can handle that ourselves where needed.

The `ViewBinder` interface requires a `setViewValue()` method where we do that work:

```
  @Override
  public boolean setViewValue(View view, Cursor cursor, int columnIndex) {
    if (columnIndex==2) {
      long time=cursor.getLong(columnIndex);
      String formattedTime=DateUtils.formatDateTime(this, time,
                           DateUtils.FORMAT_ABBREV_RELATIVE);

      ((TextView)view).setText(formattedTime);

      return(true);
    }

    return(false);
  }
```

(from Contacts/CallLog/app/src/main/java/com/commonsware/android/calllog/consumer/CallLogConsumerActivity.java)

If `setViewValue()` returns `false`, that indicates that the `SimpleCursorAdapter` should handle that column normally. Our implementation does that for everything other than column 2, which in our `PROJECTION` is the date. For the date, we get the

`long` value of the date and use `DateUtils.formatDateTime()` to convert it into a string representation that will be more human-readable. We put that string into the `TextView` and return `true` to indicate that we have handled this widget binding ourselves.

The results is a list of calls by date and phone number.

The CalendarContract Provider

The Android Open Source Project (AOSP) has had a Calendar application from its earliest days. This application originally was designed to sync with Google Calendar, later extended to other sync sources, such as Microsoft's Exchange. However, this application was not part of the Android SDK, so there was no way to access it from your Android application.

At least, no officially documented and supported way.

Many developers poked through the AOSP source code and found that the Calendar application had a `ContentProvider`. Moreover, this `ContentProvider` was exported (by default). So many developers used undocumented and unsupported means for accessing calendar information. This occasionally broke, as Google modified the Calendar app and changed these pseudo-external interfaces.

Android 4.0 added official SDK support for interacting with the Calendar application via its `ContentProvider`. As part of the SDK, these new interfaces should be fairly stable — if nothing else, they should be supported indefinitely, even if new and improved interfaces are added sometime in the future. So, if you want to tie into the user's calendars, you can. Bear in mind, though, that the new `CalendarContract ContentProvider` is not identical to the older undocumented providers, so if you are aiming to support pre-4.0 devices, you have some more work to do.

Of course, similar to the `ContactsContract ContentProvider`, the `CalendarContract ContentProvider` is severely lacking in documentation, and anything not documented is subject to change.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- content provider theory
- content provider implementations

You Can't Be a Faker

While the Android emulator has the `CalendarContract` `ContentProvider`, it will do you little good. While you can define a Google account on the emulator, the emulator lacks any ability to sync content with that account. Hence, you cannot see any events for your calendars in the Calendar app, and you cannot access any calendar data via `CalendarContract`.

You may be able to use an `outlook.com` account, to sync with an Outlook calendar.

Otherwise, in order to test your use of `CalendarContract`, you will need to have hardware that runs Android 4.0 (or higher), with one or more accounts set up that have calendar data.

Do You Have Room on Your Calendar?

As a `ContentProvider`, `CalendarContract` is not significantly different from any other such provider that Android supplies or that you write yourself, in that there are `Uri` values representing collections of data, upon which you can query, insert, update, and delete as needed.

The Collections

The two main collections of data that you are likely to be interested in are `CalendarContract.Calendars` (the collection of all defined calendars) and `CalendarContract.Events` (the collection of all defined events across all calendars). Each of those has a `CONTENT_URI` static data member that you would use with `ContentResolver` or a `CursorLoader` to perform operations on those collections. An entry in `CalendarContract.Events` points back to its corresponding calendar via a `CALENDAR_ID` column that you can query upon; the remaining columns on `CalendarContract.Events` have names apparently designed to match with the

iCalendar specification (e.g., DTSTART and DTEND for the start and end times of the event).

Three other collections may be of interest:

1. CalendarContract.Instances has one entry per occurrence of an event, so recurring events get multiple rows
2. CalendarContract.Attendees has information about each attendee of an event
3. CalendarContract.Reminders has information about each reminder scheduled for an event (e.g., when to remind the user), for those events with associated reminders

Each of those ties back to its associated CalendarContract.Events row via an EVENT_ID column.

Calendar Permissions

There are two permissions for working with CalendarContract: READ_CALENDAR and WRITE_CALENDAR. As you might expect, querying CalendarContract requires the READ_CALENDAR permission; modifying CalendarContract data requires the WRITE_CALENDAR permission.

These permissions have existed since Android's earliest days, even in the SDK, as a side effect of the "meat cleaver" approach the core Android team employed to create the initial SDK. Hence, you can request these permissions in the manifest with any Android build target, without compiler errors. Of course, actually referring to CalendarContract will require a build target (i.e., compileSdkVersion in Android Studio) of API Level 14 or higher.

Querying for Events

For example, let's populate a ListView with the roster of all events the user has across all calendars, using a CursorLoader, showing the name of each event, the event's start date, and the event's end date. You can find this in the Calendar/Query sample project in the book's source code.

Our manifest has the READ_CALENDARS permission, as you would expect:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
  package="com.commonsware.android.cal.query"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk android:minSdkVersion="14"/>

  <uses-permission android:name="android.permission.READ_CALENDAR"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".CalendarQueryActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Calendar/Query/app/src/main/AndroidManifest.xml)

We will use a simple `ListActivity` and so therefore do not need an activity layout. Our row layout (`res/layout/row.xml`) has three `TextView` widgets for the three pieces of data that we want to display:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/linearLayout1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="4dip"
    android:layout_weight="1"
    android:ellipsize="end"
    android:textSize="20sp"/>

  <LinearLayout
```

```
    android:id="@+id/linearLayout2"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_marginRight="4dip"
    android:orientation="vertical">

    <TextView
      android:id="@+id/dtstart"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:textSize="10sp"/>

    <TextView
      android:id="@+id/dtend"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="bottom"
      android:textSize="10sp"/>
  </LinearLayout>

</LinearLayout>
```

(from Calendar/Query/app/src/main/res/layout/row.xml)

In our activity (`CalendarQueryActivity`), in `onCreate()`, we set up a `SimpleCursorAdapter` on a `null` `Cursor` at the outset and define the activity as being the adapter's `ViewBinder`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    adapter=
        new SimpleCursorAdapter(this, R.layout.row, null, ROW_COLUMNS,
                                ROW_IDS);
    adapter.setViewBinder(this);
    setListAdapter(adapter);

    getLoaderManager().initLoader(0, null, this);
  }
```

(from Calendar/Query/app/src/main/java/com/commonsware/android/cal/query/CalendarQueryActivity.java)

A `ViewBinder` is a way to tailor how `Cursor` data is poured into row widgets, without subclassing the `SimpleCursorAdapter`. Implementing the `SimpleCursorAdapter.ViewBinder` interface requires us to implement a

`setViewValue()` method, which will be called when the adapter wishes to pour data from one column of a `Cursor` into one widget. We will examine this method shortly.

The `SimpleCursorAdapter` will pour data from the `ROW_COLUMNS` in our `Cursor` into the `ROW_IDS` widgets in our row layout:

```
private static final String[] ROW_COLUMNS=
    new String[] { CalendarContract.Events.TITLE,
        CalendarContract.Events.DTSTART,
        CalendarContract.Events.DTEND };
private static final int[] ROW_IDS=
    new int[] { R.id.title, R.id.dtstart, R.id.dtend };
```

(from Calendar/Query/app/src/main/java/com/commonsware/android/cal/query/CalendarQueryActivity.java)

Our `onCreate()` also initializes the `Loader` framework, triggering a call to `onCreateLoader()`, where we create and return a `CursorLoader`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(this, CalendarContract.Events.CONTENT_URI,
                          PROJECTION, null, null,
                          CalendarContract.Events.DTSTART));
}
```

(from Calendar/Query/app/src/main/java/com/commonsware/android/cal/query/CalendarQueryActivity.java)

We query on `CalendarContract.Events.CONTENT_URI`, asking for a certain set of columns indicated by our `PROJECTION` static data member:

```
private static final String[] PROJECTION=
    new String[] { CalendarContract.Events._ID,
        CalendarContract.Events.TITLE,
        CalendarContract.Events.DTSTART,
        CalendarContract.Events.DTEND };
```

(from Calendar/Query/app/src/main/java/com/commonsware/android/cal/query/CalendarQueryActivity.java)

The `ROW_COLUMNS` we map are a subset of the `PROJECTION`, skipping the `_ID` column that `SimpleCursorAdapter` needs but will not be displayed. Our query is also set up to sort by the start date (`CalendarContract.Events.DTSTART`).

When the query is complete, we pop it into the adapter in `onLoadFinished()` and remove it in `onLoaderReset()`:

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
  adapter.swapCursor(cursor);
```

```
  }

  public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
  }
```

(from Calendar/Query/app/src/main/java/com/commonsware/android/cal/query/CalendarQueryActivity.java)

Our `setViewValue()` implementation then converts the `DTSTART` and `DTEND` values into formatted strings by way of `DateUtils` and the `formatDateTime()` method:

```
  @Override
  public boolean setViewValue(View view, Cursor cursor, int columnIndex) {
    long time=0;
    String formattedTime=null;

    switch (columnIndex) {
      case 2:
      case 3:
        time=cursor.getLong(columnIndex);
        formattedTime=
            DateUtils.formatDateTime(this, time,
                                     DateUtils.FORMAT_ABBREV_RELATIVE);
        ((TextView)view).setText(formattedTime);
        break;

      default:
        return(false);
    }

    return(true);
  }
}
```

(from Calendar/Query/app/src/main/java/com/commonsware/android/cal/query/CalendarQueryActivity.java)

The `setViewValue()` method should return `true` for any columns it handles and `false` for columns it does not — skipped columns are handled by `SimpleCursorAdapter` itself.

If you run this on a device with available calendar data, you will get a list of those events:

Figure 775: The Calendar Query sample application, with some events redacted

Penciling In an Event

What is rarely documented in the Android SDK is what activities might exist that support the MIME types of a given `ContentProvider`. In part, that is because device manufacturers have the right to remove or replace many of the built-in applications.

The Calendar application is considered by Google to be a "core" application. Quoting the Android 2.3 version of the Compatibility Definition Document (CDD):

> The Android upstream project defines a number of core applications, such as a phone dialer, calendar, contacts book, music player, and so on. Device implementers MAY replace these applications with alternative versions. However, any such alternative versions MUST honor the same Intent patterns provided by the upstream project. For example, if a device contains an alternative music player, it must still honor the Intent pattern issued by third-party applications to pick a song.

Hence, in theory, so long as the CDD does not change and device manufacturers correctly honor it, those Intent patterns described by the Calendar application's

manifest should be available across Android 4.0 devices. The Calendar application appears to support `ACTION_INSERT` and `ACTION_EDIT` for both the collection MIME type (`vnd.android.cursor.dir/event`) and the instance MIME type (`vnd.android.cursor.item/event`). Notably, there is no support for `ACTION_PICK` to pick a calendar or event, the way you can use `ACTION_PICK` to pick a contact.

The MediaStore Provider

Playing back media is a popular pastime on Android devices, one in which your app may want to participate. The easiest way for you to find out what media is available for you to display, edit, or otherwise work with is via the `MediaStore` content provider. `MediaStore` is part of the Android framework and allows you to query for images, audio files, and video files that are indexed on the device.

This chapter will review the general workings of `MediaStore`, plus work through an example of getting video files — and their thumbnails — from `MediaStore`.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- content provider theory
- content provider implementations

It is also a pretty good idea to have read the chapters on media recording and playback that might be of relevance, depending on what you intend to do with the `MediaStore`:

- Audio Playback
- Audio Recording
- Video Playback
- Using the Camera via 3rd-Party Apps
- Working Directly with the Camera

You might also wish to consider skimming through the chapter on files again, as it will be cross-referenced in several places in this chapter.

What Is the MediaStore?

The documentation for `MediaStore` describes it this way:

> The Media provider contains meta data for all available media on both internal and external storage devices.

This definition... leaves a bit to be desired.

From our standpoint as Android developers, the `MediaStore` is a `ContentProvider`, supplied by Android. We can use it much like we use other system-supplied providers, like `ContactsContract` and `CalendarContract`. In this case, the primary role of `MediaStore` is for us to find media, just as the primary role of `ContactsContract` is for us to find contacts.

The "meta data" reference in the documentation refers to the fact that `MediaStore` itself does not store the media, even though that's what the name `MediaStore` would suggest. `MediaMetadataStore` would be a more accurate description. We can learn *about* available media — names, durations, etc. — and we can get a `Uri` from `MediaStore` pointing to the media, but the media itself lives as a file somewhere else.

Indexed Media

`MediaStore` has media as a primary focus. Here, "media" refers to:

- Images (typically photos)
- Audio (music, podcasts, etc.)
- Video (whether recorded by the device, downloaded from somewhere, etc.)

`MediaStore` has intrinsic knowledge of these, particularly for the file formats and codecs that Android supports. As a result, the index maintained by `MediaStore` will contain some metadata in common for all file types, such as:

- title
- MIME type
- dates (when the file was added, when the file was modified)

...and other metadata that will be unique to one or two of the major types, such as:

- duration for audio and video (but not images)

- height and width for images and video (but not audio)
- geotagging for images and video (but not audio)

Indexed Non-Media

As was mentioned in passing in the chapter on files, Android uses MTP for Android 4.0+ as the USB protocol for sharing files with a desktop or notebook computer.

To power this, Android does not go straight to the filesystem, but rather works with `MediaStore`. `MediaStore` maintains an index of *all* files, not just "media". Whatever shows up in `MediaStore` is what shows up to the user in their Windows drive letter, macOS mounted volume, etc.

You too can query `MediaStore` for non-media files. Android will try to maintain a MIME type — probably based on file extensions — and so you can find all indexed PDF files, for example, by querying `MediaStore`.

MediaStore and "Other" External Storage

In the chapter on files, we covered the difference between internal storage and external storage. Primarily, `MediaStore` maintains an index of external storage.

However, many Android devices today have multiple locations that could be considered "external storage". While the vast majority of Android devices have "external storage" as a portion of on-board flash memory, Android device manufacturers are welcome to add other options, such as:

- card slots (typically microSD)
- USB host ports (capable of mounting thumb drives and the like)

From the standpoint of the Android SDK, such secondary storage locations are off-limits, in that there is nothing in the Android SDK to tell us if there are any such locations, where they are located (in terms of `File` objects to their roots), whether they can be read from, or whether they can be written to. You will find various blog posts and Stack Overflow answers where developers have attempted to catalog all of the possibilities, using a mix of low-level Linux information and manufacturer-based heuristics, but these techniques will be generally unreliable across thousands of device models.

However, many manufacturers who have added such secondary storage options will arrange to have that storage be indexed and be part of `MediaStore`. So, if the user slides in a microSD card containing audio files, on many devices, when you query `MediaStore` for available audio files, you will find those on the microSD card in addition to those on "traditional" external storage. From the user's standpoint, in terms of consuming media, this is sufficient.

How Does My Content Get Indexed?

As was noted back in the chapter on files, if you write files to external storage, you will want to use `MediaScannerConnection` to ensure that those files get indexed. In that chapter, the focus was on ensuring that your files would be visible to attached desktops/notebooks via MTP. However, what *really* happens is that `MediaScannerConnection` updates `MediaStore`, which in turn drives the MTP-served content.

Even if you fail to index content manually, at some point, Android is likely to pick up the files. For example, Android will scan external storage after a reboot. However, using `MediaScannerConnection` to "tap Android on the shoulder" and have it index your file means that it will show up in `MediaStore` more quickly. This is very important for multimedia assets — if you downloaded some media, you want that to be indexed as soon as possible, so the user can turn around and consume that media, whether through your app or another one on the user's device.

How Do I Retrieve Video from the MediaStore?

Video players will need to find out what videos are available on the device, eligible for playback. They may wish to retrieve other details, such as the video title, duration, and so forth. And, of course, they will need something that they can use to actually play back the video itself.

In this section, we will work through the `Media/VideoList` sample project. This project has a `VideosFragment` that will show the roster of available videos; tapping on a video in the list will launch the user's video player to watch that video.

Requesting Permission

Starting on API Level 19 devices, you need to hold the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions to be able to work with the `MediaStore`.

Hence, the `VideoList` sample app has the `READ_EXTERNAL_STORAGE` permission in its manifest, as it has no need to write to external storage:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.video.list"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="17"/>

  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

  <application
    android:allowBackup="false"
    android:hardwareAccelerated="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Media/VideoList/app/src/main/AndroidManifest.xml)

Querying for Video

`VideosFragment` uses the `Loader` framework, since `MediaStore` is a `ContentProvider` and `Loader` is a convenient way to asynchronously load content from a

`ContentProvider`. `VideosFragment` implements the `LoaderManager.LoaderCallbacks` interface and, in `onActivityCreated()`, calls `getLoaderManager().initLoader()` to initialize its `Loader`.

That triggers a call to `onCreateLoader()`, where `VideosFragment` creates a `CursorLoader` to query the `MediaStore` for videos:

```
  @Override
  public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
    return(new CursorLoader(
                            getActivity(),
                            MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
                            null, null, null,
                            MediaStore.Video.Media.TITLE));
  }
```

(from Media/VideoList/app/src/main/java/com/commonsware/android/video/list/VideosFragment.java)

The `Uri` for video content from `MediaStore` is `MediaStore.Video.Media.EXTERNAL_CONTENT_URI`. Passing in `null` for the list of columns to return will return all available columns — not the most efficient approach, but it is convenient. The sort order of `MediaStore.Video.Media.TITLE` has the results sorted by the `TITLE` column, so the videos are returned alphabetically.

Back up in `onViewCreated()`, we initialized a `SimpleCursorAdapter` to handle our results, passing in the `TITLE` and `_ID` columns into our custom row layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:padding="8dp">

  <ImageView
    android:id="@+id/thumbnail"
    android:layout_width="64dp"
    android:layout_height="64dp"
    android:contentDescription="@string/thumbnail"/>

  <TextView
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_gravity="center_vertical"
```

```
    android:textSize="24sp"/>

</LinearLayout>
```

(from Media/VideoList/app/src/main/res/layout/row.xml)

onViewCreated() also attaches a custom ViewBinder, ThumbnailBinder, that we will cover in the next section, before eventually attaching the initially-empty SimpleCursorAdapter to the ListView of our ListFragment:

```
  @Override
  public void onViewCreated(View view,
                            Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    String[] from=
      { MediaStore.Video.Media.TITLE, MediaStore.Video.Media._ID };
    int[] to= { android.R.id.text1, R.id.thumbnail };
    SimpleCursorAdapter adapter=
      new SimpleCursorAdapter(getActivity(), R.layout.row, null,
        from, to, 0);

    adapter.setViewBinder(this);
    setListAdapter(adapter);

    getLoaderManager().initLoader(0, null, this);
  }
```

(from Media/VideoList/app/src/main/java/com/commonsware/android/video/list/VideosFragment.java)

The rest of our LoaderManager.LoaderCallbacks methods are fairly conventional, using swapCursor() to load in the results of the query (or null if the loader is reset):

```
  @Override
  public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
    ((CursorAdapter)getListAdapter()).swapCursor(c);
  }

  @Override
  public void onLoaderReset(Loader<Cursor> loader) {
    ((CursorAdapter)getListAdapter()).swapCursor(null);
  }
```

(from Media/VideoList/app/src/main/java/com/commonsware/android/video/list/VideosFragment.java)

Showing the Thumbnails

If you have used a video player on Android, most have an activity (or fragment) akin to the one we are implementing in this section. And, most of those will show thumbnail images of the videos in question.

However, retrieving and showing those thumbnails is a bit complicated, because Android may need to *generate* the thumbnail, if there is not already a thumbnail for the video, or if its cache of thumbnails was cleared. Generating a thumbnail takes time, time that we do not want to spend on the main application thread. For that, we will use Picasso, as profiled in the chapter on Internet access.

The thumbnail will need to be displayed using some sort of `ImageView`. Since `SimpleCursorAdapter` cannot populate an `ImageView` directly, we need some other way to fill in the `ImageView`. To handle this, we create an implementation of a `ViewBinder`, named `ThumbnailBinder` — that is what we attached to our `SimpleCursorAdapter` via `setViewBinder()` back in `onActivityCreated()`.

A `ViewBinder` is a way to tailor how `Cursor` data is poured into row widgets, without subclassing the `SimpleCursorAdapter`. Implementing the `SimpleCursorAdapter.ViewBinder` interface requires us to implement a `setViewValue()` method, which will be called when the adapter wishes to pour data from one column of a `Cursor` into one widget:

```
@Override
public boolean setViewValue(View v, Cursor c, int column) {
  if (column == c.getColumnIndex(MediaStore.Video.Media._ID)) {
    Uri video=
        ContentUris.withAppendedId(
          MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
          c.getInt(column));

    Picasso.with(getActivity()).load(video.toString())
      .fit().centerCrop()
      .placeholder(R.drawable.ic_media_video_poster)
      .into((ImageView)v);

    return(true);
  }

  return(false);
}
```

(from Media/VideoList/app/src/main/java/com/commonsware/android/video/list/VideosFragment.java)

If our `ViewBinder` is being asked to bind the video ID column (`MediaStore.Video.Media._ID`), we first construct the `Uri` to the video using `ContentUris.withAppendedId()`. Then, we create a UIL-supplied `DisplayImageOptions` object, where we can provide details for how to handle the image load. In this case, we supply a drawable resource to use as a placeholder image while the thumbnail is being loaded. Finally, we tell Picasso to display the image. Picasso recognizes the `Uri` structure for a video from `MediaStore` and uses utility methods like `getThumbnail()` on `MediaStore.Video.Thumbnails` to actually retrieve the thumbnail.

The net result is that when we populate our `ListView` with our `ViewBinder`-enhanced `SimpleCursorAdapter`, the `ListView` rows will initially have the placeholder image, replaced by the actual video thumbnails as they get loaded.

Playing the Selection

`VideosFragment` extends a version of `ContractListFragment`, as was used in the EU4You samples earlier in this book. The activity that hosts this fragment is obligated to implement the `VideosFragment.Contract` interface, which in turn requires an `onVideoSelected()` method.

In `onListItemClick()` of `VideosFragment`, the fragment calls `onVideoSelected()` on the `Contract`, supplying:

- the `String` representation of the `Uri` that points to the video itself, pulled from the `MediaStore.Video.Media.DATA` column of the `Cursor` we loaded from the `MediaStore`
- the MIME type of that video, pulled from the `MediaStore.Video.Media.MIME_TYPE` column of that same `Cursor`

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
  CursorAdapter adapter=(CursorAdapter)getListAdapter();
  Cursor c=(Cursor)adapter.getItem(position);
  int uriColumn=c.getColumnIndex(MediaStore.Video.Media.DATA);
  int mimeTypeColumn=
      c.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);

  getContract().onVideoSelected(c.getString(uriColumn),
                                c.getString(mimeTypeColumn));
}
```

(from Media/VideoList/app/src/main/java/com/commonsware/android/video/list/VideosFragment.java)

The main activity — surprisingly named `MainActivity` — loads up a `VideosFragment` as a static fragment via the `res/layout/main.xml` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/videos"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.video.list.VideosFragment"
/>
```

(from Media/VideoList/app/src/main/res/layout/main.xml)

`MainActivity` implements `VideosFragment.Contract` and therefore has an `onVideoSelected()` method. It simply constructs an `Intent` to view the video and starts an activity with it:

```
package com.commonsware.android.video.list;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import java.io.File;

public class MainActivity extends Activity implements
    VideosFragment.Contract {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  @Override
  public void onVideoSelected(String uri, String mimeType) {
    Uri video=Uri.fromFile(new File(uri));
    Intent i=new Intent(Intent.ACTION_VIEW);

    i.setDataAndType(video, mimeType);
    startActivity(i);
  }
}
```

(from Media/VideoList/app/src/main/java/com/commonsware/android/video/list/MainActivity.java)

The Results

Running this on a device with videos available should show the list of those videos, complete with title and thumbnail:

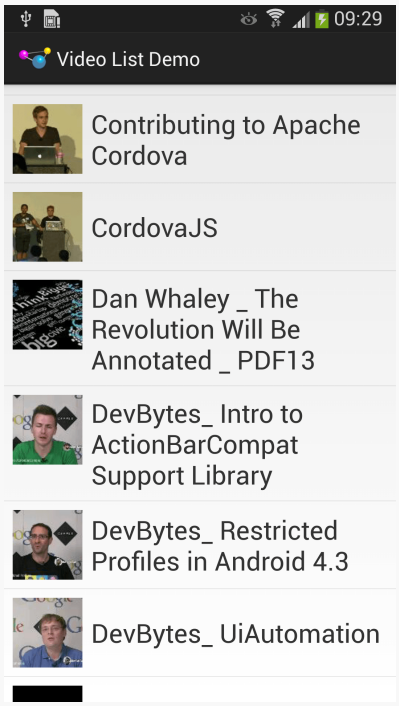


Figure 776: The Video List Demo App

Tapping on any entry in the list should bring up a video player on your device, assuming that one or more such players (that are capable of supporting `content://Uri` values) are installed.

Consuming Documents

Android has long offered the ability for an app to pick some file or stream from another app and consume it. However, the original options were designed around an app loading content from another app. Even though our *code* would be requesting content based on abstractions like MIME types, the *implementation* and user experience would be based on the traditional "pick an app to fulfill this request" chooser.

Google, given its clear interest in cross-cutting storage engines like Google Drive, wanted something better. In Android 4.4, they added the Storage Access Framework (SAF) to provide a better user experience, with only modest changes to client code. With Android's increasing reliance upon content and document providers for cross-app content sharing, understanding the Storage Access Framework is fairly important for modern app development.

In this chapter, we will examine what it takes to consume documents published via the SAF.

Prerequisites

This chapter assumes that you have read the chapter on `ContentProvider` patterns or have equivalent experience with consuming streams published by a `ContentProvider`.

The Storage Access… What?

Let's think about photos for a minute.

A person might have photos managed as:

- on-device photos, mediated by an app like a gallery
- photos stored online in a photo-specific service, like Instagram
- photos stored online in a generic file-storage service, like Google Drive or Dropbox

Now, let's suppose that person is in an app that allows the user to pick a photo, such as to attach to an email.

The classic Android solution would be for the user to have to first choose the app to use to find the photo (e.g., Gallery, Instagram, Google Drive, Dropbox), then find the photo using that app. Then, if all goes well, the original app would receive a `Uri` to that photo and be able to make use of it.

However, this flow has three main problems:

1. From the user's standpoint, they need to know where they have the photo before they can go looking for it. Given the prominence of generic file-storage services, the user might not remember where the photo is stored, but might remember enough details about the photo (e.g., timeframe when taken, tags that might have been attached to the photo) to find it... but the user has to sequentially search each possible photo-storing app until the right one is found.
2. From the client app developer's standpoint, too many apps screw up handling the classic `ACTION_PICK` and `ACTION_GET_CONTENT` activities, failing to return a result in all cases. Users then are as likely to blame the client app for the mistake as they are to blame the photo-storing app or Android itself.
3. None of this was designed with online file-sharing services in mind. What happens if an app knows about a possible file, but the file is not available on the device right now, because it has not been downloaded from the online service?

The Storage Access Framework is designed to address these issues. It provides its own "picker" UI to allow users to find a file of interest that matches the MIME type that the client app wants. File providers simply publish details about their available files — including those that may not be on the device but could be retrieved if needed. The picker UI allows for easy browsing and searching across all possible file providers, to streamline the process for the user. And, since Android is the one providing the picker, the picker should more reliably give a result to the client app based upon the user's selection (if any).

The Storage Access Framework Participants

Providers are specialized `ContentProvider` implementations, usually extending `DocumentsProvider`, that can tell Android about the documents that are published by an app. This includes providing any sort of organizational structure (directory tree, tag cloud, etc.)

The **clients** are apps that wish to consume (or create) documents managed by providers. Clients will indicate what sort of document they want, in the form of a MIME type, where applicable.

The **picker** is the system UI that allows the user to pick a document (or documents) from among the documents published by all providers that meet the criteria established by a client requesting access to the document(s).

Picking How to Pick (a Peck of Pickled Pepper Photos)

`ACTION_PICK` would seem to be the `Intent` action to use to pick something. It works, but it is designed for the case where you know the specific collection of "somethings" you want to pick from. Use this, for example, to pick a contact specifically out of `ContactsContract`.

In cases where you know the MIME type you want, but you do not particularly know or care about the exact source of the file, use `ACTION_GET_CONTENT` on API Level 18 and below for everything.

For MIME types that clearly represent a document, file, or other sort of stream, use `ACTION_OPEN_DOCUMENT` (and the SAF) on API Level 19+. The SAF picker will incorporate both full-fledged SAF-compliant providers' documents along with apps that only support `ACTION_GET_CONTENT`. However, since `ACTION_OPEN_DOCUMENT` is only available on API Level 19+ devices, if you are supporting older devices, you will need to check `Build.VERSION.SDK_INT` and choose an `Intent` action accordingly.

For MIME types that represent entries in a database (e.g., a calendar entry), use `ACTION_GET_CONTENT`, even on API Level 19+. Google also recommends using `ACTION_GET_CONTENT` on API Level 19+ "if you want your app to simply read/import data", though it is unclear why they make this recommendation or why the user experience should differ based upon how the bytes would be used.

Opening a Document

Technically, we do not "open" a document using `ACTION_OPEN_DOCUMENT`. Instead, we are requesting a `Uri` pointing to some document that the user chooses.

To do that, create an `Intent` with:

- `ACTION_OPEN_DOCUMENT` as the action
- `CATEGORY_OPENABLE` as the category
- your desired MIME type

Then, use that `Intent` with `startActivityForResult()`.

For example, the `Documents/Consumer` sample application contains a `ConsumerFragment` that adds an "Open" item to the action bar overflow. Clicking on "Open" triggers a call to the `open()` method on the fragment. And, for API Level 19+ devices, that will in turn request to "open" a document:

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private void open() {
  Intent i=new Intent().setType("image/*");

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
    startActivityForResult(i.setAction(Intent.ACTION_OPEN_DOCUMENT)
                            .addCategory(Intent.CATEGORY_OPENABLE),
                           REQUEST_OPEN);
  }
  else {
    startActivityForResult(i.setAction(Intent.ACTION_GET_CONTENT)
                            .addCategory(Intent.CATEGORY_OPENABLE),
                           REQUEST_GET);
  }
}
```

(from Documents/Consumer/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

This `open()` method also gracefully degrades for older devices, falling back to `ACTION_GET_CONTENT`. In both cases, we are trying to allow the user to pick some image (MIME type of `image/*`). The two `startActivityForResult()` calls use different request IDs (`REQUEST_OPEN` versus `REQUEST_GET`), so that we can distinguish the sort of result that we get in `onActivityResult()`:

```
@Override
public void onActivityResult(int requestCode, int resultCode,
```

```
                                Intent resultData) {
  if (resultCode==Activity.RESULT_OK) {
    if (resultData!=null) {
      uri=resultData.getData();

      if (requestCode==REQUEST_OPEN) {
        getLoaderManager().initLoader(0, null, this);
      }
      else {
        logToTranscript(uri.toString());
      }
    }
  }
}
```

(from Documents/Consumer/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

Both `ACTION_GET_CONTENT` and `ACTION_OPEN_DOCUMENT` should supply a `Uri` in the result `Intent` that points to the document the user chose, if the user actually chose one and we got `RESULT_OK` as the result code. This sample logs that `Uri` value to a "transcript" (`TextView` inside of a `ScrollView`) to show what we get back.

If the result is from an `ACTION_OPEN_DOCUMENT` request (`REQUEST_OPEN` request code), we can try to get some metadata about the document. The provider should support a query on the returned `Uri` that will give us the display name (`OpenableColumns.DISPLAY_NAME`) and possibly the size of the file (`OpenableColumns.SIZE`). So, we use the `Loader` framework to run this query, with our fragment implementing the `LoaderCallbacks`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(getActivity(), uri, PROJECTION, null, null, null));
}

public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
  transcript.setText(null);
  logToTranscript(uri.toString());

  if (c!=null && c.moveToFirst()) {
    int displayNameColumn=
      c.getColumnIndex(OpenableColumns.DISPLAY_NAME);

    if (displayNameColumn>=0) {
      logToTranscript("Display name: "
        +c.getString(displayNameColumn));
    }
```

```
      int sizeColumn=c.getColumnIndex(OpenableColumns.SIZE);

      if (sizeColumn<0 || c.isNull(sizeColumn)) {
        logToTranscript("Size not available");
      }
      else {
        logToTranscript(String.format("Size: %d",
          c.getInt(sizeColumn)));
      }
    }
    else {
      logToTranscript("...no metadata available?");
    }
  }

  public void onLoaderReset(Loader<Cursor> loader) {
    // unused
  }
```

(from Documents/Consumer/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

If we get results back, we try to read out these two values and record them to the transcript as well. Note, though, that:

- There is no guarantee that either column will be in the result set, or that the result set will have any rows
- The size might not be known, particularly if the file is not presently resident on the device (e.g., it is being downloaded now, given that the user chose the file), and so we need to call `isNull()` on the `Cursor` to see if we actually have a `SIZE` value before trying to get it as an integer

The user is presented with the system's picker, to choose an image, complete with a navigation drawer to get to various spots within the picker:

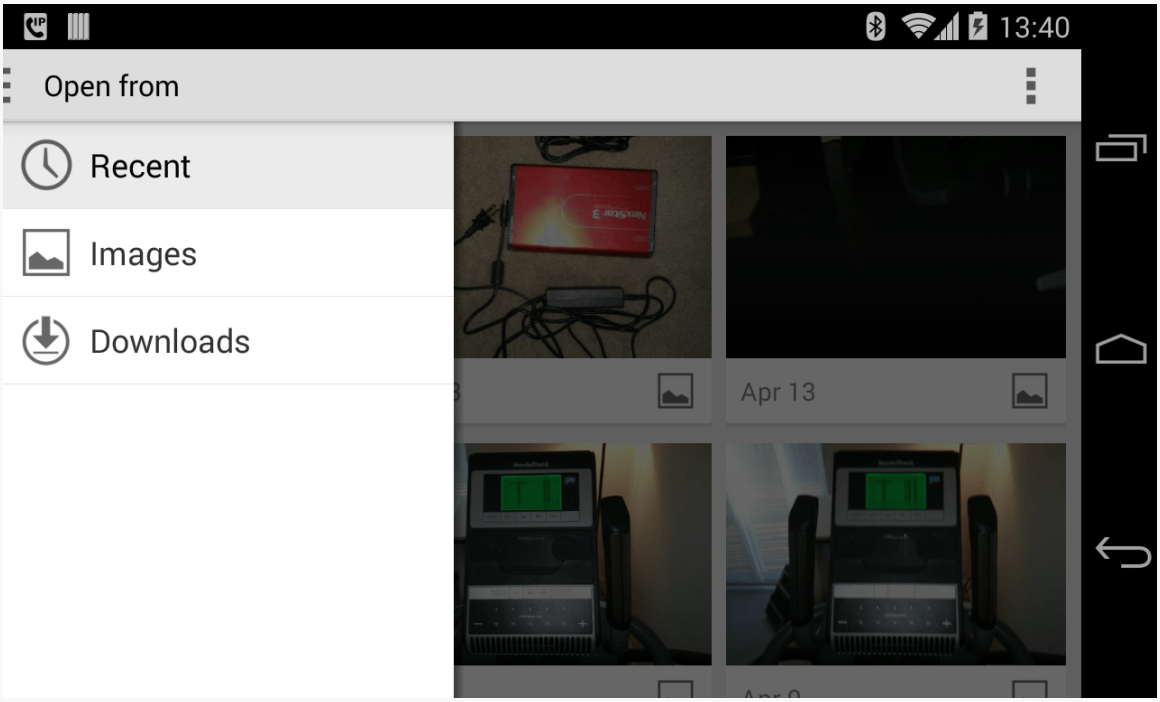


Figure 777: Storage Access Framework Picker, Showing Images

When the user taps on an image, the results wind up in our transcript UI:

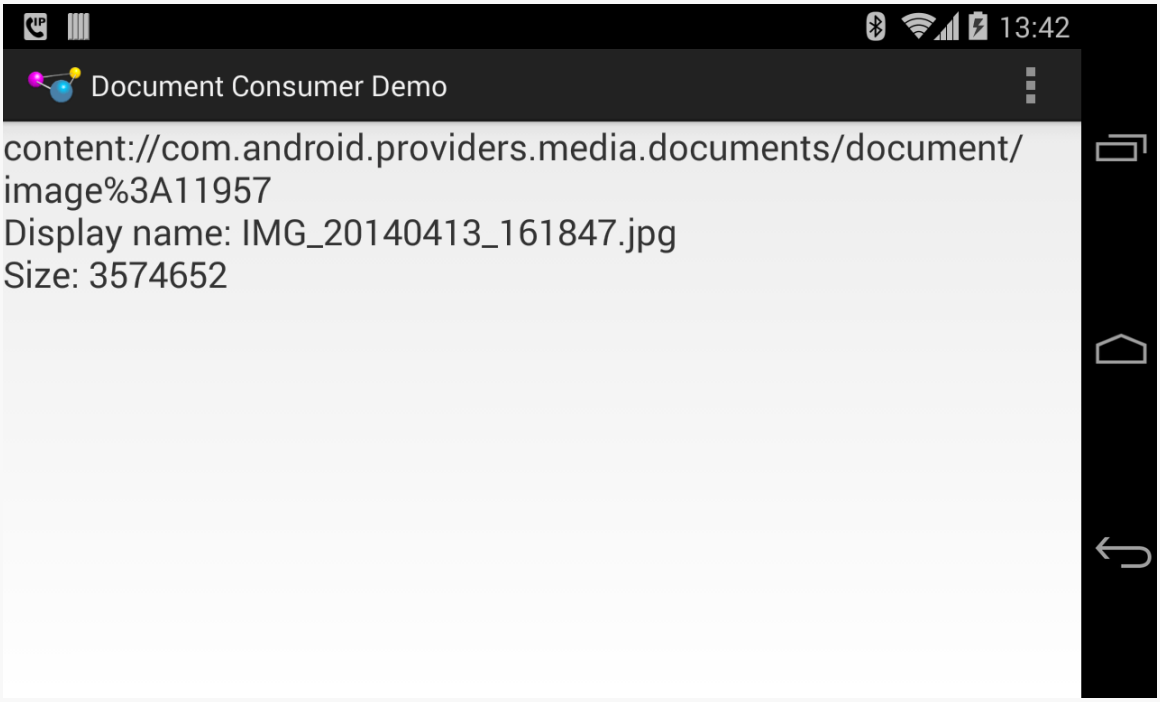


Figure 778: Uri, Display Name, and Size of Chosen File

Note that the default behavior of `ACTION_OPEN_DOCUMENT` is to let the user choose a single file. If your `Intent` includes `EXTRA_ALLOW_MULTIPLE`, set to `true`, then the user can choose multiple documents. Rather than getting their `Uri` values via `getData()` on the result `Intent`, you will need to call `getClipData()` on the `Intent` and iterate over the "clipboard entries".

The `Uri` itself can then be used to get an `InputStream` or `OutputStream` for the contents, using `openInputStream()` and `openOutputStream()` on `ContentResolver`, respectively. Note, though, that you cannot pass the `Uri` to other applications, as they may not have rights to work with that document the way that you do.

Why We Want Things To Be Openable

You will notice that both the `ACTION_OPEN_DOCUMENT` and the `ACTION_GET_CONTENT` `Intent` objects created in the preceding example have `CATEGORY_OPENABLE` applied to them. This is supposed to guarantee that we can actually consume the content represented by the `Uri` that we get. In particular, we should be able to use a `ContentResolver` and open streams on that content.

If we leave off CATEGORY_OPENABLE, it is possible that we will get a Uri that we cannot open ourselves.

The difference boils down to what use we intend to put the Uri toward:

- If all we plan to do is use that Uri for is to wrap it in an ACTION_VIEW Intent and start an activity on it, you could skip CATEGORY_OPENABLE and perhaps offer more choices to your user
- Otherwise, use CATEGORY_OPENABLE

The Rest of the CRUD

ACTION_OPEN_DOCUMENT will give you a Uri for a document that you can open for reading — the "R" in "CRUD".

However, the other CRUD operations are also entirely possible.

Create

ACTION_CREATE_DOCUMENT will give you a Uri for a document that you can open for writing, as it is your document.

To do this, construct an Intent with:

- an action of ACTION_CREATE_DOCUMENT
- a category of CATEGORY_OPENABLE
- the MIME type of the content you wish to write
- an extra, named EXTRA_TITLE, containing your desired filename

Then, invoke startActivityForResult() on that Intent, and use the Uri supplied in the result Intent delivered to onActivityResult().

We will see an example of ACTION_CREATE_DOCUMENT later in this chapter.

Update

The Uri returned from an ACTION_OPEN_DOCUMENT request may be writable. If it is, you can use openOutputStream() on a ContentResolver to write to that document. You can determine if a document is writable by examining the COLUMN_FLAGS value

returned from a `query()` on the `Uri` — if it includes `FLAG_SUPPORTS_WRITE`, you can write to the document.

We will see an example of modifying a document obtained from the Storage Access Framework later in this chapter.

Delete

Similarly, if the `COLUMN_FLAGS` value includes `FLAG_SUPPORTS_DELETE`, you can delete the document by calling the static `deleteDocument()` method on the `DocumentsContract` class, supplying a `ContentResolver` plus the `Uri` of the document to be deleted.

The DocumentFile Helper

The `support-core-utils` library from the Android Support libraries contains a `DocumentFile` class. This provides a light `File`-like layer atop the low-level API offered through a `ContentResolver` or `DocumentsContract`. Of note, `DocumentFile` offers convenience methods for:

- determining read/write access (`canRead()` and `canWrite()`)
- getting the display name (`getName()`) and length (`length()`) of the document
- getting the MIME type (`getType()`)
- deleting the document (`delete()`)
- and so on

`DocumentFile` can actually work with raw files — just create a `DocumentFile` using the `static fromFile()` method. More often, though, you will create a `DocumentFile` wrapped around the `Uri` that you get from `ACTION_OPEN_DOCUMENT` or `ACTION_CREATE_DOCUMENT`. For that, there is the `static fromSingleUri()` method.

CWAC-Document and DocumentFileCompat

There is one significant problem with `DocumentFile`, though: it only works on document `Uri` values on API Level 19 and higher. However, we can get such `Uri` values on older devices, and we can get `Uri` values from other places than a `DocumentProvider` (e.g., `FileProvider`).

The CWAC-Document library offers a `DocumentFileCompat` class. This is forked from the official `DocumentFile` but adds support for older devices and non-document `Uri` values. Specifically, for a non-document `content` `Uri`, you can get:

- the display name, queried via the `OpenableColumns`
- the length, also from the `OpenableColumns`
- the MIME type, from `ContentResolver` and `getType()`

Also, some `DocumentFile` methods whose values can be safely hard-coded for legacy documents, like `isVirtual()` (`false`), `isDirectory()` (`false`), and `isFile()` (`true`) are supported.

`DocumentFileCompat` also offers convenience methods for:

- Accessing streams (`openInputStream()`, `openOutputStream()`)
- Copying the content to something else (`copyTo()`)
- Copying something else into this content (`copyFrom()`, `copyFromAsset()`)

We will see the use of `DocumentFileCompat` in the next couple of sample apps.

Getting Durable Access

By default, you will have the rights to read (and optionally write) to the document represented by the `Uri` until the activity that requested the document via `ACTION_OPEN_DOCUMENT` or `ACTION_CREATE_DOCUMENT` is destroyed.

If you pass the `Uri` to another component — such as a service – you will need to add `FLAG_GRANT_READ_URI_PERMISSION` and/or `FLAG_GRANT_WRITE_URI_PERMISSION` to the `Intent` used to start that component. That extends your access until *that* component is destroyed.

If, however, you need the rights to survive your app restarting, you can call `takePersistableUriPermission()` on a `ContentResolver`, indicating the `Uri` of the document and the permissions (`FLAG_GRANT_READ_URI_PERMISSION` and/or `FLAG_GRANT_WRITE_URI_PERMISSION`) that you want persisted. Those rights will then survive a reboot. However:

- They will not survive the document being deleted, so just because you have a saved `Uri`, do not assume that the `Uri` will still be valid
- Not every storage provider will grant such persistable permissions

- This will only work for `Uri` values that you get from `ACTION_OPEN_DOCUMENT` or `ACTION_CREATE_DOCUMENT`, not ones that you get by other means (e.g., `ACTION_GET_CONTENT`)
- You can revoke those rights by calling `releasePersistableUriPermission()` later on, and there is nothing stopping the storage provider from revoking those permissions itself at some time in the future

In addition, you can call `getPersistedUriPermissions()` to find out what persisted permissions your app has. This returns a `List` of `UriPermission` objects, where each one of those represents a `Uri`, what persisted permissions (read or write) you have, and when the permissions will expire.

If you do not have persistable permissions when you get the `Uri`, your primary recourse is to make a local copy of the content, while you still have temporary access to it. The `Documents/Durable` sample application illustrates this. It is based on the `Documents/Consumer` sample app from earlier in this chapter, with a few key changes:

- It provides menu options for opening a document via `ACTION_OPEN_DOCUMENT` or getting content via `ACTION_GET_CONTENT`
- It uses an `IntentService` (`DurablizerService`) to either obtain persistable permissions to the content or to make a local copy, delivering the results asynchronously via an event bus
- It uses `DocumentFileCompat` instead of direct `ContentResolver` queries to get the content metadata

For an event bus, we use greenrobot's EventBus, added as a dependency via `build.gradle`, along with the CWAC-Document library:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 25
        applicationId "com.commonsware.android.documents.durable"
    }
}

repositories {
```

```
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation 'com.commonsware.cwac:document:0.2.0'
}
```

(from Documents/Durable/app/build.gradle)

`MainActivity` is the same as in the `Documents/Consumer` sample app, as is the core UI of `ConsumerFragment` (`ScrollView` with a `TextView` inside). However, `ConsumerFragment` registers for events in `onStart()` and unregisters in `onStop()`:

```
  @Override
  public void onStart() {
    super.onStart();

    EventBus.getDefault().register(this);
  }

  @Override
  public void onStop() {
    EventBus.getDefault().unregister(this);

    super.onStop();
  }
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

The menu resource now has two items, `open` (for `ACTION_OPEN_DOCUMENT`) and `get` (for `ACTION_GET_CONTENT`):

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/get"
    android:showAsAction="never"
    android:title="@string/get" />
  <item
    android:id="@+id/open"
    android:enabled="false"
    android:showAsAction="never"
    android:title="@string/open" />
```

```
</menu>
```

(from Documents/Durable/app/src/main/res/menu/actions.xml)

The open one is disabled (`android:enabled="false"`) at the outset, as `ACTION_OPEN_DOCUMENT` only works on API Level 19+ devices, and the `minSdkVersion` of this sample is 15.

In `onCreateOptionsMenu()`, we conditionally enable the open item, and in `onOptionsItemSelected()`, we route the menu items to `open()` and `get()` methods:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.actions, menu);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.KITKAT) {
    menu.findItem(R.id.open).setEnabled(true);
  }

  super.onCreateOptionsMenu(menu, inflater);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.open) {
    open();
  }
  else if (item.getItemId()==R.id.get) {
    get();
  }

  return(super.onOptionsItemSelected(item));
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

Those methods invoke their associated `Intent` actions, using `*/*` this time for the MIME type:

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private void open() {
  Intent i=
    new Intent()
      .setType("*/*")
      .setAction(Intent.ACTION_OPEN_DOCUMENT)
      .addCategory(Intent.CATEGORY_OPENABLE);
```

```
  startActivityForResult(i, REQUEST_OPEN);
}

private void get() {
  Intent i=
    new Intent()
      .setType("image/png")
      .setAction(Intent.ACTION_GET_CONTENT)
      .addCategory(Intent.CATEGORY_OPENABLE);

  startActivityForResult(i, REQUEST_GET);
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

However, whereas the `Documents/Consumer` sample app processed the resulting `Uri` directly in `onActivityResult()`, now we delegate that work to a `DurablizerService` that is responsible for ensuring that we have durable access to the content:

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                             Intent resultData) {
  if (resultCode==Activity.RESULT_OK) {
    getActivity()
      .startService(new Intent(getActivity(), DurablizerService.class)
        .setData(resultData.getData())
        .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION));
  }
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

We pass the `Uri` via the data facet of the `Intent` (`setData()`), plus add the `FLAG_GRANT_READ_URI_PERMISSION` flag. This ensures that `DurablizerService` will have access to the content even if `MainActivity` is destroyed before our work is complete.

`onHandleIntent()` in `DurablizerService` has a basic flow:

- Try to obtain persistable permissions
- If that does not work, try to make a local copy
- If either of those work, deliver a `DocumentFile` to the UI layer via a `ContentReadyEvent`

```
@Override
protected void onHandleIntent(Intent intent) {
  Uri document=intent.getData();
  boolean weHaveDurablePermission=obtainDurablePermission(document);

  if (!weHaveDurablePermission) {
    document=makeLocalCopy(document);
  }

  if (weHaveDurablePermission || document!=null) {
    Log.d(getClass().getSimpleName(), document.toString());

    DocumentFileCompat docFile=buildDocFileForUri(document);

    Log.d(getClass().getSimpleName(),
      "Display name: "+docFile.getName());
    Log.d(getClass().getSimpleName(),
      "Size: "+Long.toString(docFile.length()));

    EventBus.getDefault().post(new ContentReadyEvent(docFile));
  }
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/DurablizerService.java)

The job of `obtainDurablePermission()` is to use `takePersistableUriPermission()` on a `ContentResolver` to request read and write access, at least on Android 4.4 or higher. So, we create a `perms` value that requests both `FLAG_GRANT_READ_URI_PERMISSION` and `FLAG_GRANT_WRITE_URI_PERMISSION`, and pass that to `takePersistableUriPermission()`, along with the `document` `Uri`:

```
private boolean obtainDurablePermission(Uri document) {
  boolean weHaveDurablePermission=false;

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.KITKAT) {
    int perms=Intent.FLAG_GRANT_READ_URI_PERMISSION
      | Intent.FLAG_GRANT_WRITE_URI_PERMISSION;

    try {
      getContentResolver()
        .takePersistableUriPermission(document, perms);

      for (UriPermission perm :
        getContentResolver().getPersistedUriPermissions()) {
        if (perm.getUri().equals(document)) {
          weHaveDurablePermission=true;
        }
```

```
      }
    }
    catch (SecurityException e) {
      // OK, we were not offered any persistable permissions
    }
  }

  return(weHaveDurablePermission);
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/DurablizerService.java)

Unfortunately, while `takePersistableUriPermission()` is synchronous, it does not actually tell us if we *have* those permissions. The only way to find out if we have access is to call `getPersistedUriPermissions()`, which returns a roster of `UriPermission` objects listing *every* persisted `Uri` permission held by our app. Then, we have to sift through those, looking for one matching our desired document `Uri`.

`obtainDurablePermission()` then returns a `boolean` indicating whether or not our request for persistable permissions succeeded.

`makeLocalCopy()` will be called if `obtainDurablePermission()` returns `false` (e.g., we are on an older Android device or otherwise used `ACTION_GET_CONTENT`). Its job is to make a local copy of the content, so we have indefinite access:

```
private Uri makeLocalCopy(Uri document) {
  DocumentFileCompat docFile=buildDocFileForUri(document);
  Uri result=null;

  if (docFile.getName()!=null) {
    try {
      String ext=
        MimeTypeMap.getSingleton().getExtensionFromMimeType(docFile.getType());

      if (ext!=null) {
        ext="."+ext;
      }

      File f=File.createTempFile("cw_", ext, getFilesDir());

      docFile.copyTo(f);
      result=Uri.fromFile(f);
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
        "Exception copying content to file", e);
    }
  }

  return(result);
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/DurablizerService.java)

We start off by getting a `DocumentFileCompat` for the `document` `Uri`, using a `buildDocFileForUri()` helper method:

```
private DocumentFileCompat buildDocFileForUri(Uri document) {
  DocumentFileCompat docFile;

  if (document.getScheme().equals(ContentResolver.SCHEME_CONTENT)) {
    docFile=DocumentFileCompat.fromSingleUri(this, document);
  }
  else {
    docFile=DocumentFileCompat.fromFile(new File(document.getPath()));
  }

  return(docFile);
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/DurablizerService.java)

This handles both likely scenarios:

- We get a `content:` `Uri`, and so our `DocumentFileCompat` needs to be created using the `fromSingleUri()` factory method
- We get a `file:` `Uri`, and so our `DocumentFileCompat` needs to be created using the `fromFile()` factory method

Back in `makeLocalCopy()`, we get the file extension associated with the content's MIME type, by way of `MimeTypeMap` and `getExtensionFromMimeType()`. Calling `getType()` on the `DocumentFileCompat` will fail if, for some reason, we have a `content:` `Uri` but do not have any permission to work with it. For example, had we failed to include `FLAG_GRANT_READ_URI_PERMISSION` on the `Intent` that started `DurablizerService` *and* if the `MainActivity` were destroyed by this point, we would not have any rights to use the content, and `getType()` will fail to get the MIME type for the content.

But, if `getType()` succeeds, we try to get that file extension. That too might fail, returning `null`, if the MIME type is not recognized by `MimeTypeMap`. We can live with a `null` file extension. However, if the file extension is *not* `null`, it will lack the leading `.`, so we add that. Then, we create a unique file in `getFilesDir()`, using that file extension and `createTempFile()` on `File`. From there, we use the convenient `copyTo()` method on `DocumentFileCompat` to copy that content into our temporary file.

The `ContentReadyEvent` simply wraps a `DocumentFileCompat` pointing to our final content: either the original `Uri` (if we obtained durable access to it) or on our local copy:

```
static class ContentReadyEvent {
  final DocumentFileCompat docFile;

  ContentReadyEvent(DocumentFileCompat docFile) {
    this.docFile=docFile;
  }
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/DurablizerService.java)

Back in `ConsumerFragment`, when the `ContentReadyEvent` is received, we log the details to the transcript:

```
@Subscribe(threadMode=ThreadMode.MAIN)
public void onContentReady(DurablizerService.ContentReadyEvent event) {
  logToTranscript(event.docFile.getUri().toString());
  logToTranscript("Display name: "+event.docFile.getName());
  logToTranscript("Size: "+Long.toString(event.docFile.length()));
}
```

(from Documents/Durable/app/src/main/java/com/commonsware/android/documents/consumer/ConsumerFragment.java)

For cases where we are granted persistable permissions, the output will show a `content: Uri`, as we can continue to use the original content:

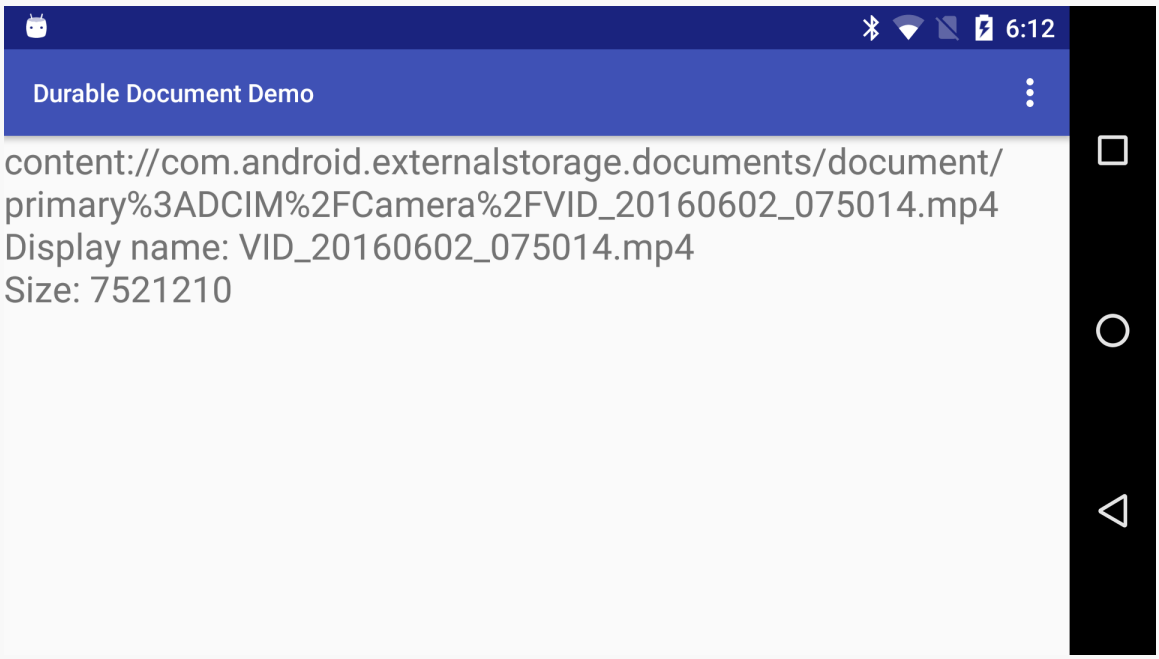


Figure 779: Durable Document Demo, Showing Document Result

For cases where we were not granted persistable permissions — such as `ACTION_GET_CONTENT` — the output will show a `file:` `Uri`, representing the local copy of the content:

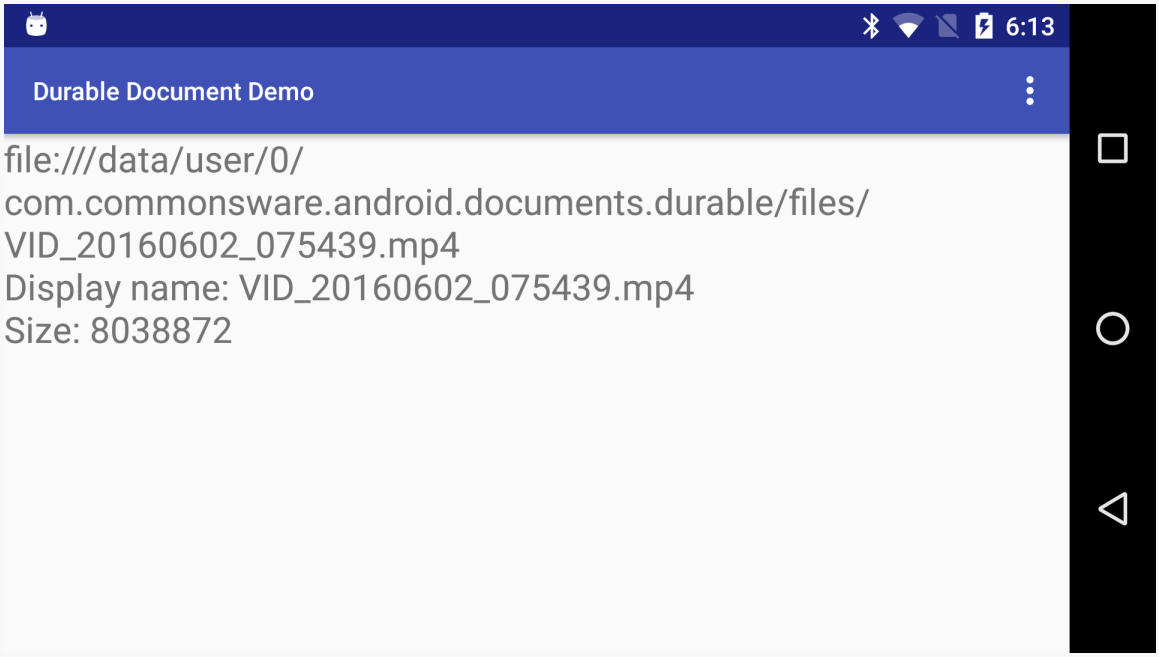


Figure 780: Durable Document Demo, Showing Content Result

Another Durable Example: Diceware

Dealing with durable documents does not require a service. Services are a good idea when you might need to make a copy of the document (e.g., the `ACTION_GET_CONTENT` scenario) and you do not know how big that document might be. In cases where the document is known to be small, just using background threads is fine, such as via RxJava.

With that in mind, here is another sample app that demonstrates using `ACTION_OPEN_DOCUMENT` and `ACTION_GET_CONTENT` to work with user-provided documents. Unlike most examples in this book, though, this app has some direct usefulness, as it is a passphrase generator, using the diceware approach.

Dice? Where?!?

Simply put, diceware is a technique for generating a lengthy passphrase that still has a chance of being memorable, by choosing words from a list of common words.

The name “diceware” comes from the canonical way of generating the passphrase: rolling five six-sided dice (the sorts of dice that you see in casinos and board games)

and looking up words in a word list. The more randomly-chosen words in the passphrase, the stronger the passphrase and the more time it would take for it to be cracked by somebody.

Perhaps the most famous such passphrase is `correct horse battery staple`, though a four-word passphrase is probably a bit short today.

We Want Words!

To use the diceware approach to generate a passphrase, you need a list of words.

A traditional diceware word list contains two columns. The right-hand column is the word, and the left-hand column is the dice roll that corresponds to that word:

```
4244  liverwurst
4245  lizard
4246  llama
4251  luau
4252  lubricant
4253  lucidity
4254  ludicrous
4255  luggage
```

The canonical diceware site has a number of word lists for a variety of languages. Each of those lists has 7,776 words, which equates to 6^5, or the number of combinations of five rolls of a six-sided die.

However, other word lists also exist. The EFF has published a few lists, aiming for more commonly-used words and eliminating ones that might cause confusion (e.g., homophones). They also have some short lists, with only 1,296 (6^4) words, with an emphasis on shorter words. Shorter word lists require more words in the passphrase to get the same level of security (e.g., 8 words from the short lists result in similar security as 6 words from the longer lists).

The `Documents/Diceware` sample application packages one of the EFF word lists in the app itself, as an asset, so the app is usable "out of the box" to generate a passphrase. However, it also has action bar items to allow you to use `ACTION_OPEN_DOCUMENT` or `ACTION_GET_CONTENT` to use a different word list, if you wanted.

The Results

First, let's take a look at what the app does, before we see how it is built.

When you run the app, you immediately get a `CardView` showing a randomly-generated passphrase, using the word list that is baked into the app:

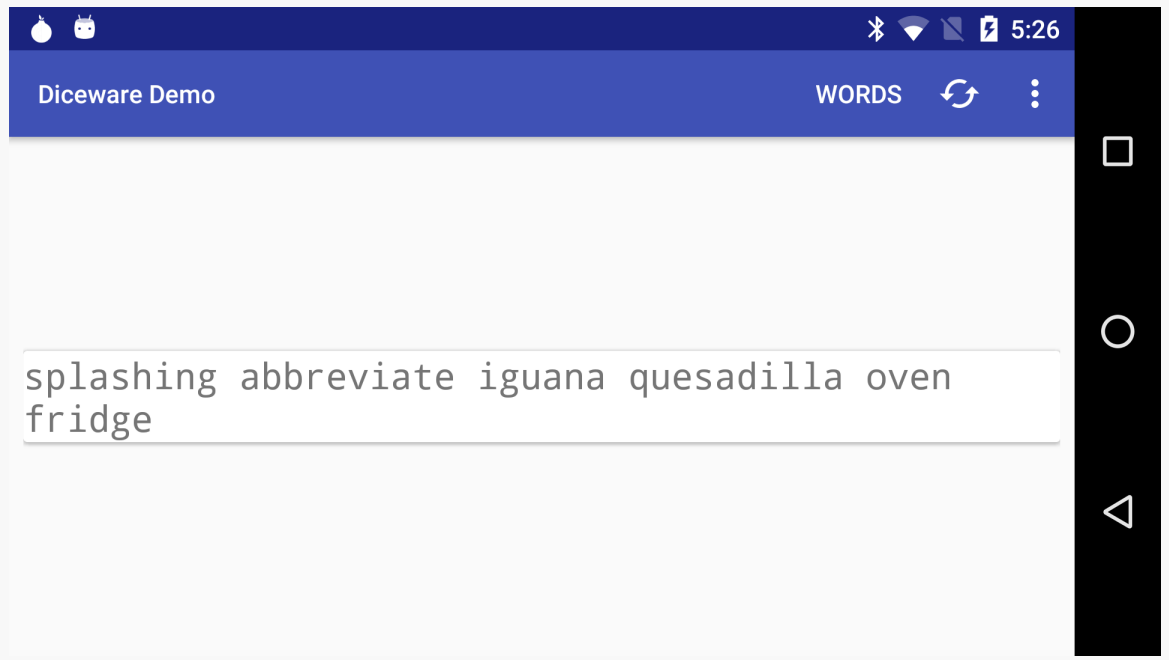


Figure 781: Diceware App, As Initially Launched

Clicking the refresh action bar item will generate a fresh passphrase, while tapping the "Words" action bar item lets you choose the length of the passphrase (from 4-10 words):

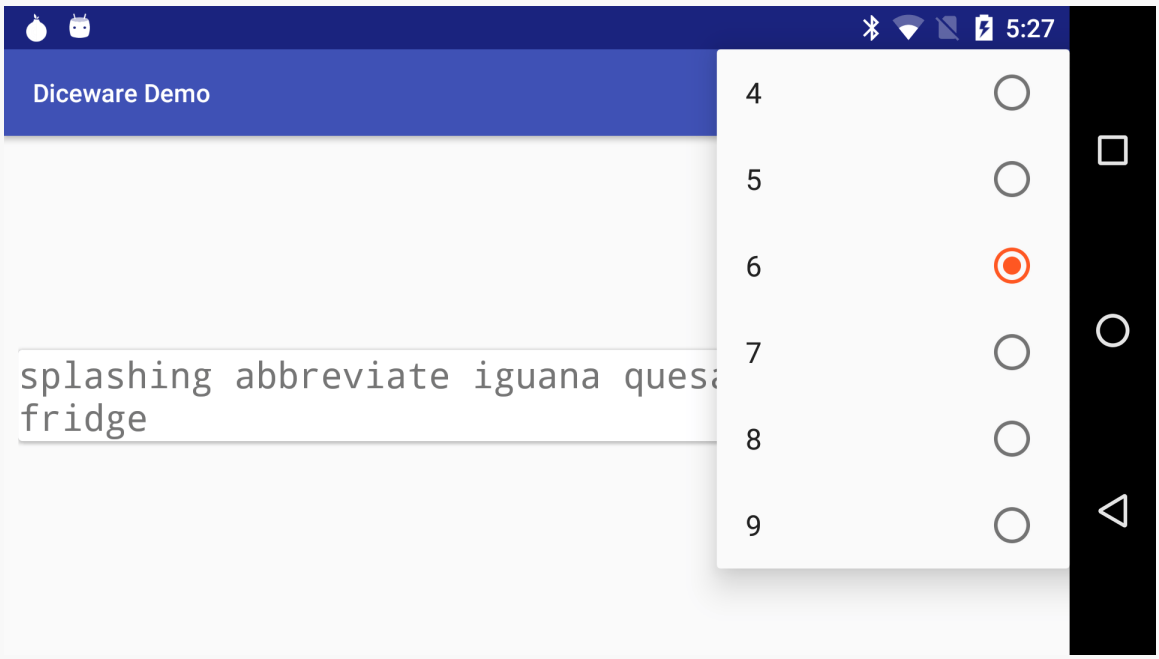


Figure 782: Diceware App, Showing Word Count Submenu

There are three options in the action bar overflow:

- "Get Word File", which uses `ACTION_GET_CONTENT` to allow you to pick an alternative word list file
- "Open Word File", which uses `ACTION_OPEN_DOCUMENT` to allow you to do the same
- "Reset", which switches you back to the built-in word list

How We Got There

The `Diceware` app consists of a single `PassphraseFragment`, loaded using a `FragmentTransaction` by the `MainActivity`.

Loading Our Words

`PassphraseFragment` uses two RxJava `Observable` objects:

- One manages loading the list of words and randomly choosing a word from that list (`wordsObservable`)
- The other manages getting durable access to the external word file that you choose to load in (`docObservable`)

When the app first runs, neither of these are set up yet. In `onViewCreated()`, we confirm that we do not have a `docObservable` at the moment — if we did, that would indicate that we are still in the middle of getting durable access to some words. Normally, `docObservable` will be `null` here, and so we call a `loadWords()` method:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  passphrase=view.findViewById(R.id.passphrase);

  if (savedInstanceState!=null) {
    wordCount=savedInstanceState.getInt(STATE_WORD_COUNT);
  }

  if (docObservable!=null) {
    docSub();
  }
  else {
    loadWords(false, wordsObservable==null);
  }
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

(we will cover `docObservable`, `docSub()`, and `wordCount` later)

`loadWords()` takes two `boolean` parameters:

- The first is "should we forcibly reload our words, even if we already loaded them before?"
- The second is "do we want to generate a new passphrase?"

`PassphraseFragment` is a retained fragment. It is entirely possible that we already have our words loaded from disk, in which case we do not need to load them again (e.g., after the user rotated the screen). So, here we pass `false` for the first parameter.

However, just because the user rotated the screen does not mean that we want to generate a fresh passphrase. We only want to do that if we do not already have a passphrase, or have one being generated right now. If `wordsObservable` exists (is not `null`), then we know that a passphrase has either been displayed or is soon to be displayed, and so we can skip creating a new one. When the app is first launched,

though, `wordsObservable` is `null`, and so we tell `loadWords()` to generate a passphrase.

`loadWords()` then sets up `wordsObservable`:

```
private void loadWords(boolean forceReload, boolean regenPassphrase) {
  if (wordsObservable==null || forceReload) {
    final Application app=getActivity().getApplication();

    wordsObservable=Observable
      .defer(() -> (Observable.just(PreferenceManager
        .getDefaultSharedPreferences(app))))
      .subscribeOn(Schedulers.io())
      .map(sharedPreferences -> {
        PassphraseFragment.this.prefs=sharedPreferences;

        return(sharedPreferences.getString(PREF_URI, ""));
      })
      .map(s -> {
        InputStream in;

        if (s.length()==0) {
          in=app.getAssets().open(ASSET_FILENAME);
        }
        else {
          in=app.getContentResolver().openInputStream(Uri.parse(s));
        }

        return(readWords(in));
      })
      .cache()
      .observeOn(AndroidSchedulers.mainThread());
  }

  unsubWords();

  if (regenPassphrase) {
    wordsSub=wordsObservable.subscribe(this::rollDemBones, error -> {
        Toast
          .makeText(getActivity(), error.getMessage(), Toast.LENGTH_LONG)
          .show();
        Log.e(getClass().getSimpleName(), "Exception processing request",
          error);
    });
  }
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

We only need to set up `wordsObservable` if it does not already exist or `forceReload` is `true`. In either of those cases, we:

- Use `defer()` to set up an `Observable` that starts by loading the default `SharedPreferences` for our app, as that is where we will record the `Uri` of any external word list that we are supposed to be using
- Use `subscribeOn()` to move all this I/O to a background thread
- Use `map()` to extract the `String` preference value, keyed by `PREF_URI`, that is the `Uri` for our external word list, returning the empty string if there is no such `Uri` (note: we cannot return `null` because RxJava does not like that)
- Use another `map()` to create an `InputStream` either on that external word list or the one in `assets/`, then read in the words from that stream via `readWords()`
- Use `cache()` to get RxJava to hold onto the results of all that work, so long as we have the same `wordsObservable` object
- Use `observeOn()` to arrange to observe the results on the main application thread

`readWords()` simply reads in the lines from the asset or external word list, divides each line on the whitespace between the two columns, and extracts the word from the second column:

```
private static List<String> readWords(InputStream in) throws IOException {
  InputStreamReader isr=new InputStreamReader(in);
  BufferedReader reader=new BufferedReader(isr);
  String line;
  List<String> result=new ArrayList<>();

  while ((line = reader.readLine())!=null) {
    String[] pieces=line.split("\s");

    if (pieces.length==2) {
      result.add(pieces[1]);
    }
  }

  return(result);
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

`loadWords()` then calls `unsubWords()`. This will `dispose()` any existing subscription to `wordsObservable`:

```
private void unsubWords() {
  if (wordsSub!=null && !wordsSub.isDisposed()) {
    wordsSub.dispose();
  }
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

Then, if we need a passphrase (`regenPassphrase` is `true`), we `subscribe()` to the `wordsObservable`, routing the word list to a `rollDemBones()` method, and displaying a `Toast` and logging to Logcat in case there is some sort of error (e.g., `IOException` reading in the words).

`rollDemBones()` uses `SecureRandom` to pick `wordCount` words from the list and updates the `passphrase` `TextView` with those words:

```
private void rollDemBones(List<String> words) {
  StringBuilder buf=new StringBuilder();
  int size=words.size();

  for (int i=0;i<wordCount;i++) {
    if (buf.length()>0) {
      buf.append(' ');
    }

    buf.append(words.get(random.nextInt(size)));
  }

  passphrase.setText(buf.toString());
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

The net result, after all of that, is that when we first run the app, we create and subscribe to the `wordsObservable`, and after a brief bit of I/O, we show the initial passphrase to the user.

Getting More Words

Handling the `ACTION_GET_CONTENT` and `ACTION_OPEN_DOCUMENT` scenarios is reminiscent of the `Documents/Durable` sample from before.

The two action bar items ("Get Word File", "Open Word File") route to `get()` and `open()` methods, each of which call `startActivityForResult()` with the appropriate action string:

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private void open() {
  Intent i=
    new Intent()
      .setType("text/plain")
      .setAction(Intent.ACTION_OPEN_DOCUMENT)
      .addCategory(Intent.CATEGORY_OPENABLE);

  startActivityForResult(i, REQUEST_OPEN);
}

private void get() {
  Intent i=
    new Intent()
      .setType("text/plain")
      .setAction(Intent.ACTION_GET_CONTENT)
      .addCategory(Intent.CATEGORY_OPENABLE);

  startActivityForResult(i, REQUEST_GET);
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

In `onActivityResult()`, if we received a `RESULT_OK` response, we set up `docObservable`, where we:

- Use `defer()` and `just()` to "observe" the results of calling `createDurableContent()`, supplying it the `Intent` that is the result of the `startActivityForResult()` call
- Use `subscribeOn()` to move that work to a background thread
- Use `cache()` to cache the results for as long as we have `docObservable`
- Use `observeOn()` to observe the results on the main application thread.

`createDurableContent()`, along with `obtainDurablePermissions()`, `makeLocalCopy()`, and `buildDocFileForUri()`, does the same work that the `DurablizerService` did in the `Documents/Durable` sample app:

- Get the persistable `Uri` permissions, if we can have them
- Make a local copy of the content, if we cannot

The biggest differences are that `createDurableContent()` saves the `Uri` of either the opened document (if we got permission) or the local copy in `SharedPreferences`, and it returns a `DocumentFileCompat` representing the external word list:

```
private DocumentFileCompat createDurableContent(Intent result) throws IOException {
  Uri document=result.getData();
  ContentResolver resolver=getActivity().getContentResolver();
  boolean weHaveDurablePermission=obtainDurablePermission(resolver, document);

  if (!weHaveDurablePermission) {
    document=makeLocalCopy(getActivity(), resolver, document);
  }

  if (weHaveDurablePermission || document!=null) {
    prefs
      .edit()
      .putString(PREF_URI, document.toString())
      .commit();

    return(buildDocFileForUri(getActivity(), document));
  }

  throw new IllegalStateException("Could not get durable permission or make copy");
}

private static boolean obtainDurablePermission(ContentResolver resolver,
                                               Uri document) {
  boolean weHaveDurablePermission=false;

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.KITKAT) {
    int perms=Intent.FLAG_GRANT_READ_URI_PERMISSION
      | Intent.FLAG_GRANT_WRITE_URI_PERMISSION;

    try {
      resolver.takePersistableUriPermission(document, perms);

      for (UriPermission perm : resolver.getPersistedUriPermissions()) {
        if (perm.getUri().equals(document)) {
          weHaveDurablePermission=true;
        }
      }
    }
    catch (SecurityException e) {
      // OK, we were not offered any persistable permissions
    }
  }

  return(weHaveDurablePermission);
}

private static Uri makeLocalCopy(Context ctxt, ContentResolver resolver,
                                 Uri document)
  throws IOException {
  DocumentFileCompat docFile=buildDocFileForUri(ctxt, document);
  Uri result=null;

  if (docFile.getName()!=null) {
    String ext=
```

```
      MimeTypeMap.getSingleton().getExtensionFromMimeType(docFile.getType());

    if (ext!=null) {
      ext="."+ext;
    }

    File f=File.createTempFile("cw_", ext, ctxt.getFilesDir());

    docFile.copyTo(f);
    result=Uri.fromFile(f);
  }

  return(result);
}

private static DocumentFileCompat buildDocFileForUri(Context ctxt, Uri document) {
  DocumentFileCompat docFile;

  if (document.getScheme().equals(ContentResolver.SCHEME_CONTENT)) {
    docFile=DocumentFileCompat.fromSingleUri(ctxt, document);
  }
  else {
    docFile=DocumentFileCompat.fromFile(new File(document.getPath()));
  }

  return(docFile);
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

`onActivityResult()` then calls `docSub()`:

```
private void docSub() {
  docSub=docObservable.subscribe(documentFile -> {
    docObservable=null;
    loadWords(true, true);
  });
}
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

Here, we subscribe to `docObservable`, and when the work is complete, we set `docObservable` to `null` (indicating that we are done with it), then call `loadWords(true, true)` to reload the words from our new source and generate a new passphrase based on those words.

The net result is that tapping either of those action bar items brings up the appropriate system UI to pick a piece of content, after which we start using that content. And, since we are persisting the `Uri` to that external word list in `SharedPreferences`, we will continue using that word source for the foreseeable future.

Other Fiddly Bits

That "forseeable future" may not be all that long. If the user taps the "Reset" action bar overflow item, we `clear()` our `SharedPreferences`, then call `loadWords(true, true)` to read in the asset's words plus generate a passphrase based that word list:

```
      case R.id.reset:
        prefs.edit().clear().apply();
        loadWords(true, true);
        return(true);
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

The action bar itself is populated from a menu resource, one that defines a submenu for the possible `wordCount` values:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/word_count"
    android:showAsAction="ifRoom"
    android:title="@string/menu_words">
    <menu>
      <group android:checkableBehavior="single">
        <item
          android:id="@+id/word_count_4"
          android:title="4" />
        <item
          android:id="@+id/word_count_5"
          android:title="5" />
        <item
          android:id="@+id/word_count_6"
          android:checked="true"
          android:title="6" />
        <item
          android:id="@+id/word_count_7"
          android:title="7" />
        <item
          android:id="@+id/word_count_8"
          android:title="8" />
        <item
          android:id="@+id/word_count_9"
          android:title="9" />
        <item
          android:id="@+id/word_count_10"
          android:title="10" />
```

```
      </group>
    </menu>
  </item>
  <item
    android:id="@+id/refresh"
    android:icon="@drawable/ic_cached_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@string/menu_refresh" />
  <item
    android:id="@+id/get"
    android:showAsAction="never"
    android:title="@string/get" />
  <item
    android:id="@+id/open"
    android:enabled="false"
    android:showAsAction="never"
    android:title="@string/open" />
  <item
    android:id="@+id/reset"
    android:showAsAction="never"
    android:title="@string/menu_reset" />

</menu>
```

(from Documents/Diceware/app/src/main/res/menu/actions.xml)

We start with the `word_count_6` item checked, and the `wordCount` field is also initialized to 6. If the user taps on any of those submenu items, we update the submenu to check the proper item, figure out the new `wordCount` (by cheating and parsing the menu title as an `Integer`), and if the user changed word count values we generate a fresh passphrase (without re-loading the word list):

```
      case R.id.word_count_4:
      case R.id.word_count_5:
      case R.id.word_count_6:
      case R.id.word_count_7:
      case R.id.word_count_8:
      case R.id.word_count_9:
      case R.id.word_count_10:
        item.setChecked(!item.isChecked());

        int temp=Integer.parseInt(item.getTitle().toString());

        if (temp!=wordCount) {
          wordCount=temp;
          loadWords(false, true);
        }
```

```
        return(true);
```

(from Documents/Diceware/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

Since that `wordCount` value can change, we save it as part of the saved instance state `Bundle` in `onSaveInstanceState()`, and we pull that value back out of that `Bundle` in `onCreateView()`.

However, we are *not* putting the generated passphrase in the saved instance state `Bundle` ourselves, even though clearly that is being generated at runtime. Here, `Diceware` takes advantage of a bit of a trick. On the layout resource, the `TextView` that shows the passphrase has `android:freezesText="true"`:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="8dp">

  <android.support.v7.widget.CardView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:padding="8dp">

    <TextView
      android:id="@+id/passphrase"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:freezesText="true"
      android:textSize="20sp"
      android:typeface="monospace" />
  </android.support.v7.widget.CardView>

</FrameLayout>
```

(from Documents/Diceware/app/src/main/res/layout/activity_main.xml)

Ordinarily, the contents of a `TextView` are not part of the saved instance state `Bundle`, because Android assumes that those values are fixed (e.g., via `android:text` in the layout). In our case, we are generating the passphrase at runtime, and so `android:freezesText="true"` tells Android to hold onto our `TextView` content in the saved instance state `Bundle` automatically, the way that it does for `EditText`. Hence, with that one attribute, Android will take care of holding onto the passphrase across configuration changes for us.

Extended Example: A Tiny Text Editor

The Documents/TinyTextEditor sample application offers a more extended look at how the Storage Access Framework can be used in practice. It implements a tab-based text editor, where you use ACTION_OPEN_DOCUMENT and ACTION_CREATE_DOCUMENT to open existing text files or create new ones. Or, you can use some other app to start an ACTION_EDIT Intent on a file: or content: Uri, and have TinyTextEditor open up an editor on that content.

The actual editing is a simple multi-line EditText widget, with the results saved on demand, when the activity moves to the background, or when a given tab is closed. As a bonus, on Android 7.0 devices, you can take a tab and launch it into a separate window, for side-by-side editing.

In this sample, we will see how the Storage Access Framework can be used in a somewhat-closer-to-production setting. Along the way, we will explore some advanced ViewPager manipulations, supporting multiple windows on Android 7.0 devices, and more.

The Overall Model

Documents are represented as Uri values, obtained from the Storage Access Framework. While we retrieve some bits of metadata, like the display name, along with the text of the document, that information is only held in widgets managed by an EditorFragment. There is no TextDocument model object. For a more sophisticated app, having a clear model layer is usually a good idea.

Where we *do* have a more structured model comes in the form of edit history. The app tracks what documents are open, so the app can re-open those documents when the app's process starts up again the next time. This is not tied strictly to saved instance state, as we want to open the existing files even if the saved instance state is lost (e.g., the user leaves the app for an hour). This is in line with how many other tabbed text editors work for desktop operating systems, such as the Sublime Text editor that the author is using to write this book.

The EditHistory class manages our list of open documents, persisting them to SharedPreferences. Since SharedPreferences does not support any sort of ordered collection of strings, EditHistory maintains a single String preference, holding a JSON serialization of an ArrayList of Uri values.

Opening a Document

From the user's standpoint, a document is opened either by choosing an "open" action bar item, or by launching the app after having left it previously with open tabs. Technically, we have a third scenario: launching a separate copy of the activity to use in a separate window. While the user may not perceive that as "opening a document", from a programming standpoint it amounts to the same thing, as our old activities and fragments are for the original window, not the new window.

The Startup Flow

The app's launcher activity is `MainActivity`. It uses a `ViewPager` along with a Material Design-style tab implementation, described elsewhere in the book:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:app="http://schemas.android.com/apk/res-auto"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">

  <io.karim.MaterialTabs
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="48dp"
    app:mtIndicatorColor="@color/accent"
    app:mtSameWeightTabs="true"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>
</LinearLayout>
```

(from Documents/TinyTextEditor/app/src/main/res/layout/main.xml)

In `onCreate()`, mostly we set up the `ViewPager` and tabs, applying an `EditorsAdapter` that we will examine shortly:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```

```
    mustRestoreHistory=(savedInstanceState==null &&
            !ACTION_NEW_WINDOW.equals(getIntent().getAction()));

    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new EditorsAdapter(getFragmentManager());
    pager.setAdapter(adapter);

    tabs=(MaterialTabs)findViewById(R.id.tabs);
    tabs.setViewPager(pager);
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

The `mustRestoreHistory` value indicates whether, later on in setting up the activity, we need to restore the editor tabs from the last time the user was in the app. There are four main scenarios here:

1. We are launching a fresh activity instance (`savedInstanceState` is `null`) from the home screen launcher (our `Intent` has no `Uri`). In this case, we should restore the tabs from the last time the user was in the app.
2. We underwent a configuration change (`savedInstanceState` is not `null`). In this case, the implementation of `ViewPager` and `FragmentManager` will set up our tabs for us automatically, and so we do not need to do that ourselves.
3. We received a `Uri` in the `Intent` that started the activity, and that `Uri` is for a custom action (`ACTION_NEW_WINDOW`) that we are using. This indicates that we are being started as a separate window, and the `Uri` is the document to be opened in that window. While we will need to open that particular document, we do not need to restore other prior tabs — they are open in the original activity. We will explore the multi-window scenario more later in this chapter.
4. We received a `Uri` in the `Intent` that started the activity, and that `Uri` is for some other action. Presumably, that is `ACTION_EDIT`. In this case, if our activity is being created, we should restore our history *and* open the new document, much the way tabbed editors in desktop operating systems work.

Our `onStart()` and `onStop()` methods mostly hook us up to the greenrobot `EventBus`, which this app uses for communication from background threads to the UI layer:

```
@Override
public void onStart() {
  super.onStart();

  EventBus.getDefault().register(this);
```

```
  if (editHistory.initialize(this)) {
    loadEditors();
  }
}

@Override
public void onStop() {
  EventBus.getDefault().unregister(this);

  super.onStop();
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

We will explore the `editHistory` bit of code later in this chapter. For the moment, take it on faith that `initialize()` returns `true` if our edit history is already loaded (so we can now open our tabs, if needed), via a `loadEditors()` method.

That `loadEditors()` method, if `mustRestoreHistory` is `true`, asks the `editHistory` for the `Uri` values that we had opened the last time the user was in the app. We then iterate over that list and call `openEditor()` for each:

```
private void loadEditors() {
  if (mustRestoreHistory) {
    List<Uri> openEditors=editHistory.getOpenEditors();

    for (Uri uri : openEditors) {
      openEditor(uri);
    }

    mustRestoreHistory=false;
  }

  if (getIntent().getData()!=null) {
    openEditor(getIntent().getData());
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

If, however, `mustRestoreHistory` is `false`, but we were passed a `Uri` in our `Intent`, we open an editor on that particular document. That `Uri` would either be for a new window or for an `ACTION_EDIT` request.

The EditorsAdapter

But before we can understand what `openEditor()` does, we need to spend a few moments on `EditorsAdapter`. We associated it with the `ViewPager`, so clearly it is some implementation of `PagerAdapter`. In particular, it is a subclass of `ArrayPagerAdapter`, the author's custom `PagerAdapter` implementation. `EditorsAdapter` coordinates a series of `EditorFragment` instances, one per document being edited:

```
package com.commonsware.android.tte;

import android.app.FragmentManager;
import android.net.Uri;
import com.commonsware.cwac.pager.ArrayPagerAdapter;
import com.commonsware.cwac.pager.PageDescriptor;
import com.commonsware.cwac.pager.SimplePageDescriptor;
import java.util.ArrayList;

public class EditorsAdapter extends ArrayPagerAdapter<EditorFragment> {
  public EditorsAdapter(FragmentManager fm) {
    super(fm, new ArrayList<PageDescriptor>());
  }

  @Override
  protected EditorFragment createFragment(PageDescriptor desc) {
    Uri document=Uri.parse(desc.getFragmentTag());

    return(EditorFragment.newInstance(document));
  }

  void addDocument(Uri document) {
    add(new SimplePageDescriptor(document.toString(),
      document.getLastPathSegment()));
  }

  void updateTitle(Uri document, String title) {
    int position=getPositionForDocument(document);

    if (position>=0) {
      SimplePageDescriptor desc=
        (SimplePageDescriptor)getPageDescriptor(position);

      desc.setTitle(title);
    }
  }
```

```
  void remove(Uri document) {
    int position=getPositionForDocument(document);

    if (position>=0) {
      remove(position);
    }
  }

  int getPositionForDocument(Uri document) {
    return(getPositionForTag(document.toString()));
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorsAdapter.java)

Work on `ArrayPagerAdapter` has been discontinued. Suffice it to say that `ArrayPagerAdapter` works with a `List` of `PageDescriptor` objects, where each of those represents a single page. Those objects hold onto the title for the page (for use by our tabs) and a unique tag (used by `FragmentManager`).

In our case, the tag is the `String` representation of the `Uri`, as that is guaranteed to be unique, since we are only going to allow one open tab per document, where the document is represented by the `Uri`.

Of note for now:

- The `EditorsAdapter` constructor starts off with an empty roster of pages, supplying an empty `ArrayList` to `ArrayPagerAdapter`. We will add (and remove) pages on the fly, based upon the documents to be edited.
- The required `createFragment()` method is given a `PageDescriptor` and needs to create the fragment for that page. Here, we get the tag from the descriptor, convert that back into a `Uri`, and pass that to a `newInstance()` factory method on `EditorFragment`.
- `addDocument()` is responsible for adding a new page to the `ViewPager`, to represent a document to be edited. We chain to the `ArrayPagerAdapter` implementation of `add()`, supplying a `SimplePageDescriptor`, which is a trivial implementation of the `PageDescriptor` interface. We provide the tag (the `String` representation of the document `Uri`) and a temporary title (the last path segment off of the document `Uri`, hoping that perhaps it is some sort of filename).
- `getPositionForDocument()` calls `getPositionForTag()` on `ArrayPagerAdapter`, given the tag generated from the document `Uri`. This

will return `-1` if there is no page with this tag, or the 0-based position of the page in the `ViewPager` if we already have a page.

We will look at the remaining methods on `EditorsAdapter` later in this chapter.

Opening an Editor

With that in mind, the primary job of `openEditor()` is to call that `addDocument()` method on the `EditorsAdapter`, to open up an `EditorFragment` on the associated document:

```
private void openEditor(Uri document) {
  if (ContentResolver.SCHEME_CONTENT.equals(document.getScheme()) ||
    canWriteFiles()) {
    int position=adapter.getPositionForDocument(document);

    if (position==-1) {
      adapter.addDocument(document);
      pager.setCurrentItem(adapter.getCount()-1);

      if (!editHistory.addOpenEditor(document)) {
        Toast
          .makeText(this, R.string.msg_save_history,
            Toast.LENGTH_LONG)
          .show();
      }
    }
    else {
      pager.setCurrentItem(position);
    }
  }
  else if (ContentResolver.SCHEME_FILE.equals(document.getScheme())) {
    pendingFiles.add(document);
    ActivityCompat.requestPermissions(this, PERMS_FILE,
      REQUEST_PERMS_FILE);
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

However, this gets complicated.

There are three possibilities for the document:

- It is a `content:` `Uri`. Responsibility for getting read/write access to that `Uri` is in a `DocumentStorageService`, since that can be done without user intervention.
- It is a `file:` `Uri`, but the user has already granted us permission to use `WRITE_EXTERNAL_STORAGE`, as indicated by the `canWriteFiles()` method:

```
private boolean canWriteFiles() {
  return(ContextCompat.checkSelfPermission(this, WRITE_EXTERNAL_STORAGE)==
    PackageManager.PERMISSION_GRANTED);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

- It is a `file:` `Uri`, but we do not have permission to work with external storage yet, such as on the first run of the app.

In the first two cases, we can go ahead and start working on opening the document. In the third case (our scheme is `file` but `canWriteFiles()` returned `false`), we use `ActivityCompat.requestPermissions()` to try to get permission, adding our document to a `pendingFiles` list. Our corresponding `onRequestPermissionResult()` method confirms that we now have access and, if so, opens the pending files by calling back into `openEditor()`:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       String[] permissions,
                                       int[] grantResults) {
  super.onRequestPermissionsResult(requestCode, permissions,
    grantResults);

  if (REQUEST_PERMS_FILE==requestCode) {
    if (canWriteFiles()) {
      for (Uri document : pendingFiles) {
        openEditor(document);
      }

      pendingFiles.clear();
    }
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

If we appear to have the ability to work with the document, we check to see if we already have the document open, by calling `getPositionForDocument()` on the

EditorsAdapter. If we do have it open already, we just make that position be the current item in the ViewPager. Otherwise, we:

- Call addDocument(), as mentioned
- Call setCurrentItem() on the ViewPager to move to the newly-added page (which, courtesy of how add() works on ArrayPagerAdapter, will be added to the end of the list of pages)
- Make sure that this document is part of our edit history, which we will explore more later in this chapter

Setting Up the Fragment

At this point, we have added a document to the EditorsAdapter, triggering ViewPager and ArrayPagerAdapter to ask EditorsAdapter to create an EditorFragment to represent that page. EditorsAdapter uses a newInstance() factory method on EditorFragment, which stuffs the document Uri into the arguments Bundle, so we retain it across configuration changes and such:

```
static EditorFragment newInstance(Uri document) {
  EditorFragment frag=new EditorFragment();
  Bundle args=new Bundle();

  args.putParcelable(KEY_DOCUMENT, document);
  frag.setArguments(args);

  return(frag);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

onCreate() makes this a retained fragment (to help optimize the configuration-change scenario) and requests to have an options menu. We also register for the EventBus here, unregistering it on onDestroy():

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setHasOptionsMenu(true);
  setRetainInstance(true);
  EventBus.getDefault().register(this);
}

@Override
```

```
  public void onStop() {
    save();

    super.onStop();
  }

  @Override
  public void onDestroy() {
    EventBus.getDefault().unregister(this);

    super.onDestroy();
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

In `onStop()`, though, we call a `save()` method that we will examine later, that saves any user changes to the document, so that we do not lose them (e.g., if our process gets terminated while we are in the background).

Our editor's layout mostly consists of a large `EditText` widget, though we also have a `ProgressBar` to show if it takes a while to load the document:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <EditText
    android:id="@+id/editor"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="left|top"
    android:inputType="textMultiLine"
    android:visibility="gone" />

  <ProgressBar
    android:id="@+id/progress"
    style="?android:attr/progressBarStyleLarge"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"/>

</FrameLayout>
```

(from Documents/TinyTextEditor/app/src/main/res/layout/editor.xml)

In `onCreateView()`, we inflate the layout and grab our widgets:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);

  editor=(EditText)result.findViewById(R.id.editor);
  progress=result.findViewById(R.id.progress);

  return(result);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

In `onViewCreated()`, if our editor is empty, we assume that we have not yet loaded the document, so we ask a `DocumentStorageService` to `loadDocument()`:

```
@Override
public void onViewCreated(View view,
                          Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  if (editor.getText().length()==0) {
    DocumentStorageService.loadDocument(getActivity(),
      getDocumentUri());
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

This will happen asynchronously; in the meantime, the user sees the `ProgressBar` (the `EditText` has `android:visibility="gone"`).

Loading the Content

`DocumentStorageService` is an `IntentService`, one responsible for all I/O related to the Storage Access Framework.

Rather than having clients directly start the service, though, `DocumentStorageService` offers `static` methods that handle that bit of work, taking relevant parameters, putting them in `Intent` extras, and calling `startService()`. This helps with long-term maintenance: if the rules for `DocumentStorageService` change (e.g., different data types allowed), the `static` methods can hide those implementation details.

So, `loadDocument()` creates an explicit `Intent` for the service, attaches an action string to identify that we are going to load a document, and puts the `Uri` of the document as the "data" of the `Intent`:

```
public static void loadDocument(Context ctxt, Uri document) {
  Intent i=new Intent(ctxt, DocumentStorageService.class)
    .setAction(Intent.ACTION_OPEN_DOCUMENT)
    .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    .setData(document);

  ctxt.startService(i);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/DocumentStorageService.java)

`onHandleIntent()` is responsible for unpacking the `Intent` and delegating the real work to a dedicated method. In the case of loading the document, a `load()` method will handle that work:

```
@Override
protected void onHandleIntent(Intent intent) {
  if (Intent.ACTION_OPEN_DOCUMENT.equals(intent.getAction())) {
    load(intent.getData());
  }
  else if (Intent.ACTION_EDIT.equals(intent.getAction())) {
    save(intent.getData(),
      intent.getStringExtra(Intent.EXTRA_TEXT),
      intent.getBooleanExtra(EXTRA_CLOSING, false));
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/DocumentStorageService.java)

(we will examine the `ACTION_EDIT` path, for saving a document, later in the chapter)

The `load()` method is responsible for doing three main things:

1. Obtaining (or refreshing) our persistable `Uri` permissions, and confirming that we do indeed have access to this document
2. Loading the text of the document itself
3. Obtaining other necessary bits of metadata, such as the display name of the document and whether or not we have write access

This... gets complicated:

```
private void load(Uri document) {
  try {
    boolean weHavePermission=false;
    boolean isContent=
      ContentResolver.SCHEME_CONTENT.equals(document.getScheme());

    if (isContent) {
      int perms=Intent.FLAG_GRANT_READ_URI_PERMISSION
        | Intent.FLAG_GRANT_WRITE_URI_PERMISSION;

      getContentResolver()
        .takePersistableUriPermission(document, perms);

      for (UriPermission perm :
        getContentResolver().getPersistedUriPermissions()) {
        if (perm.getUri().equals(document)) {
          weHavePermission=true;
        }
      }
    }
    else {
      weHavePermission=true;
    }

    if (weHavePermission) {
      try {
        InputStream is=
          getContentResolver().openInputStream(document);

        try {
          String text=slurp(is);
          DocumentFile docFile;

          if (isContent) {
            docFile=DocumentFile.fromSingleUri(this, document);
          }
          else {
            docFile=DocumentFile.fromFile(new File(document.getPath()));
          }

          EventBus
            .getDefault()
            .post(
              new DocumentLoadedEvent(document, text,
                docFile.getName(), docFile.canWrite()));
        }
        finally {
          is.close();
```

```
        }
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
          "Exception loading "+document.toString(), e);
        EventBus
          .getDefault()
          .post(new DocumentLoadErrorEvent(document, e));
      }
    }
    else {
      Log.e(getClass().getSimpleName(),
        "We failed to get permissions for "+document.toString());
      EventBus
        .getDefault()
        .post(new DocumentPermissionFailureEvent(document));
    }
  }
  catch (SecurityException e) {
    Log.e(getClass().getSimpleName(),
      "Exception getting permissions for "+document.toString(), e);
    EventBus
      .getDefault()
      .post(new DocumentPermissionFailureEvent(document));
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/DocumentStorageService.java)

We check to see what the scheme is of the document `Uri`. If it is a file, we should already have permission through the runtime permission system from the UI layer, so we can go ahead.

If, however, it is a `content` `Uri`, we need to take (or refresh) our persistable permissions for that `Uri`. So, we create a `perms` value that indicates that we want both read and write permissions to the `Uri`, then call `takePersistableUriPermission()` on a `ContentResolver` to request that access. We then check the `getPersistedUriPermissions()` output to see if we actually got our desired permissions.

If we appear to have permission, we then use `openInputStream()` on a `ContentResolver` to read in the text, leveraging a `slurp()` static method to just read in all the text from the `InputStream` and return the result:

```
// based on http://stackoverflow.com/a/309718/115145
```

```
private static String slurp(final InputStream is)
  throws IOException {
  final char[] buffer=new char[8192];
  final StringBuilder out=new StringBuilder();
  final Reader in=new InputStreamReader(is, "UTF-8");
  int rsz=in.read(buffer, 0, buffer.length);

  while (rsz>0) {
    out.append(buffer, 0, rsz);
    rsz=in.read(buffer, 0, buffer.length);
  }

  return(out.toString());
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/DocumentStorageService.java)

We then create a `DocumentFile` for this document, based on its `Uri`. `DocumentFile`, unfortunately, does not recognize any `Uri` scheme other than `content`. So, we route to `fromFile()` or `fromSingleUri()` based on the scheme.

We use that `DocumentFile` to get the display name (`getName()`) and write access (`canWrite()`), passing those and the text to a `DocumentLoadedEvent` that we publish on the event bus. If we run into some problem, we raise either a `DocumentPermissionFailureEvent` (if the problem appears to be related to permissions) or a `DocumentLoadErrorEvent` (for other sorts of errors).

Hence, the result of `load()` should be an event, of one of those three types, posted on the event bus.

Handling the DocumentLoadedEvent

Our `EditorFragment` handles the `DocumentLoadedEvent`:

```
@Subscribe(threadMode=ThreadMode.MAIN)
public void onDocumentLoaded(DocumentStorageService.DocumentLoadedEvent event) {
  if (event.document.equals(getDocumentUri())) {
    editor.setText(event.text);
    editor.setVisibility(View.VISIBLE);
    progress.setVisibility(View.GONE);
    ((Contract)getActivity())
      .applyDisplayName(getDocumentUri(), event.displayName);
    isLoaded=true;
    canWrite=event.canWrite;

    if (!canWrite) {
      editor.setEnabled(false);
    }
```

```
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

Note that we are using a 3.x generation of greenrobot's EventBus, and so we use the `@Subscribe` annotation to indicate that we want to receive this event on the main application thread.

Since the events are "generic", and we may have multiple `EditorFragment` instances on multiple documents, we need to confirm that this event is for this fragment's particular document, by comparing `Uri` values. If it is a match, we:

- Update the `EditText` with the text
- Toggle the widget visibilities, to show the `EditText` and hide the `ProgressBar`
- Via a `Contract` interface, we ask our hosting activity to update its UI to reflect the document's actual display name
- Make note that we have loaded the document, setting `isLoaded` to `true`
- Make not of whether we can write the document, by updating a `canWrite` field
- If we cannot write to the document, disable the `EditText` to indicate that fact

`MainActivity`, when called with `applyDisplayName()`, tells the `EditorsAdapter` to update the title for a tab, then tells the tabs to reload:

```
@Override
public void applyDisplayName(Uri document,
                             String displayName) {
  adapter.updateTitle(document, displayName);
  tabs.notifyDataSetChanged();
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

`updateTitle()` on `EditorsAdapter` finds the position for the supplied document, gets the `SimplePageDescriptor` for that position, and updates the title in it:

```
void updateTitle(Uri document, String title) {
  int position=getPositionForDocument(document);

  if (position>=0) {
    SimplePageDescriptor desc=
      (SimplePageDescriptor)getPageDescriptor(position);
```

```
    desc.setTitle(title);
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorsAdapter.java)

The Open Action Bar Item

By this time, you may already be tired of opening documents. However, while much of that code will be reusable, the code path specifically was for re-opening any tabs that were open when we last left this activity, or for opening a `Uri` delivered to us by an `ACTION_EDIT` request.

To open a new document, `MainActivity` has an action bar item, named `open`, with a "folder" icon:

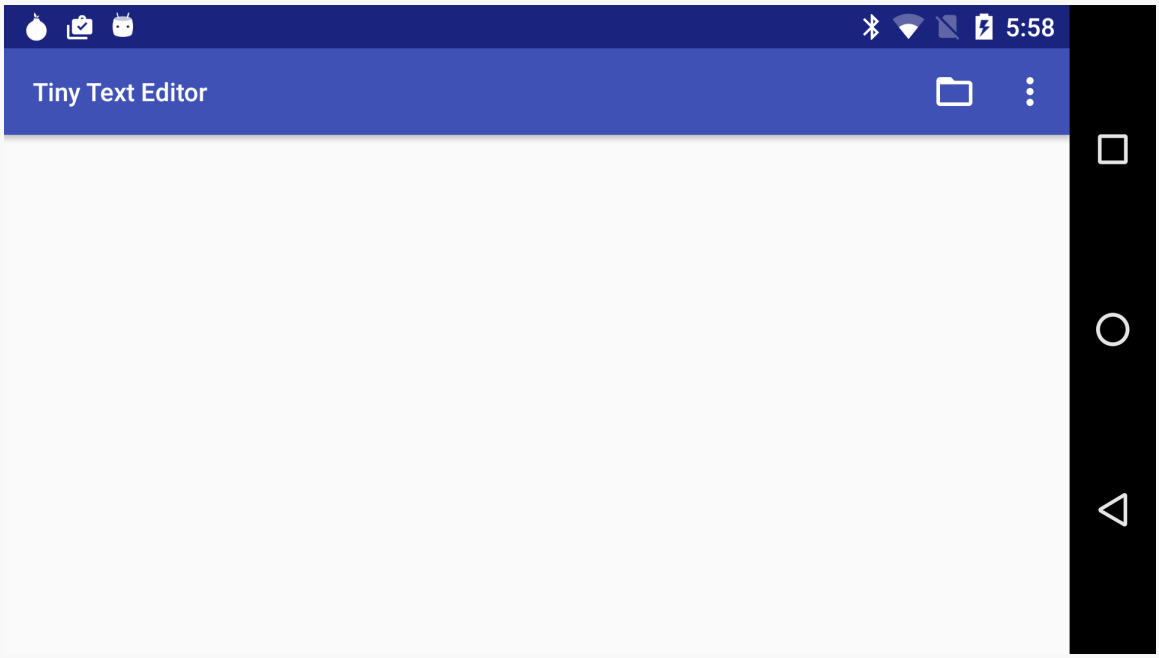


Figure 783: TinyTextEditor, Showing Open Action Bar Item

When the user taps that, our `onOptionsItemSelected()` method will call an `openDocument()` method:

```
private void openDocument(boolean allowMultiple) {
  Intent i=new Intent()
    .setType("text/*")
```

```
      .setAction(Intent.ACTION_OPEN_DOCUMENT)
      .putExtra(Intent.EXTRA_ALLOW_MULTIPLE, allowMultiple)
      .addCategory(Intent.CATEGORY_OPENABLE);

    startActivityForResult(i, REQUEST_OPEN);
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

There is also an "Open Multiple" action bar item, tossed into the overflow menu. When tapped, this too calls `openDocument()`, passing in `true` as a parameter, whereas the regular "Open" action bar item triggered an `openDocument()` call with `false`.

That flag gets packaged into the `ACTION_OPEN_DOCUMENT` `Intent` as the `EXTRA_ALLOW_MULTIPLE` value, to indicate if we want to allow the user to pick multiple files or not. This `Intent` also requests any text documents (MIME type of `text/*`).

This leads to the Storage Access Framework document picker UI, eventually triggering an `onActivityResult()` call:

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
  switch(requestCode) {
    case REQUEST_OPEN:
      if (resultCode==Activity.RESULT_OK) {
        if (data.getData()==null) {
          ClipData clip=data.getClipData();

          for (int i=0;i<clip.getItemCount();i++) {
            openEditor(clip.getItemAt(i).getUri());
          }
        }
        else {
          openEditor(data.getData());
        }
      }
      break;

    case REQUEST_CREATE:
      if (resultCode==Activity.RESULT_OK) {
        openEditor(data.getData());
      }
      break;
```

```
    }
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

For our open-document request, there are two possibilities:

- If `getData()` on the `Intent` is `null`, then we should have received multiple documents, obtained via `getClipData()` and iterating over the individual items
- Otherwise, we received a single document, whose `Uri` is the `getData()` value itself

We call `openEditor()` for the `Uri` value (or values) that we received, going through much of the same code that was shown for loading up the editor initially with past-viewed documents.

From the user's standpoint, they see the requested document in the editor:

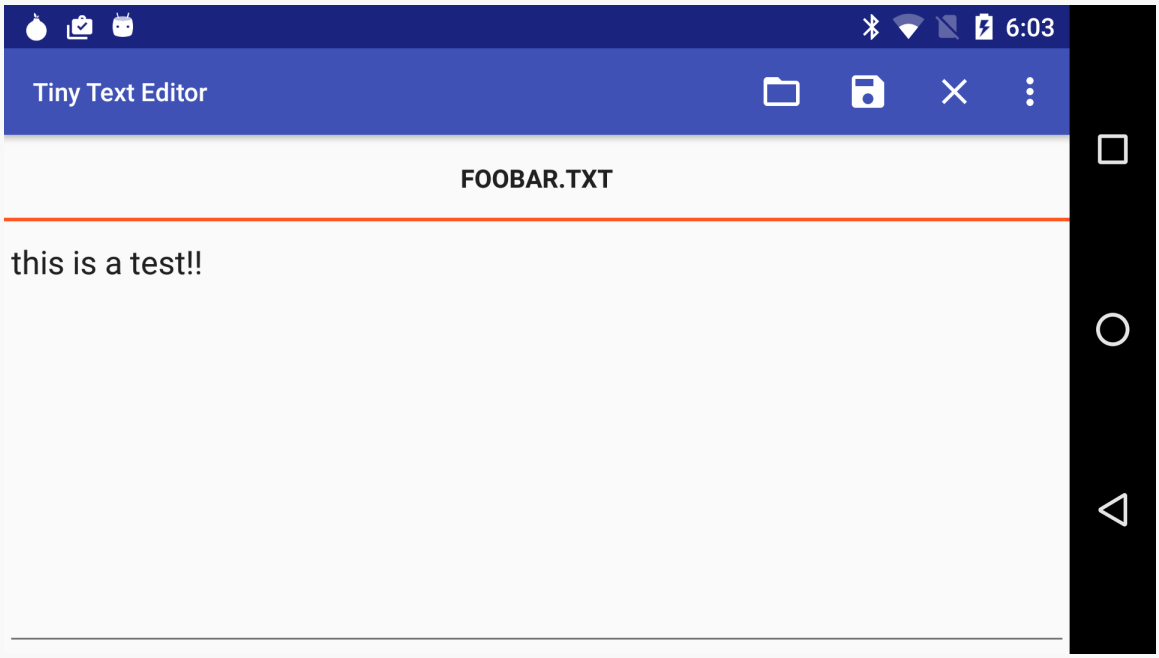


Figure 784: TinyTextEditor, Showing Opened Document

...or more than one, if they so choose:

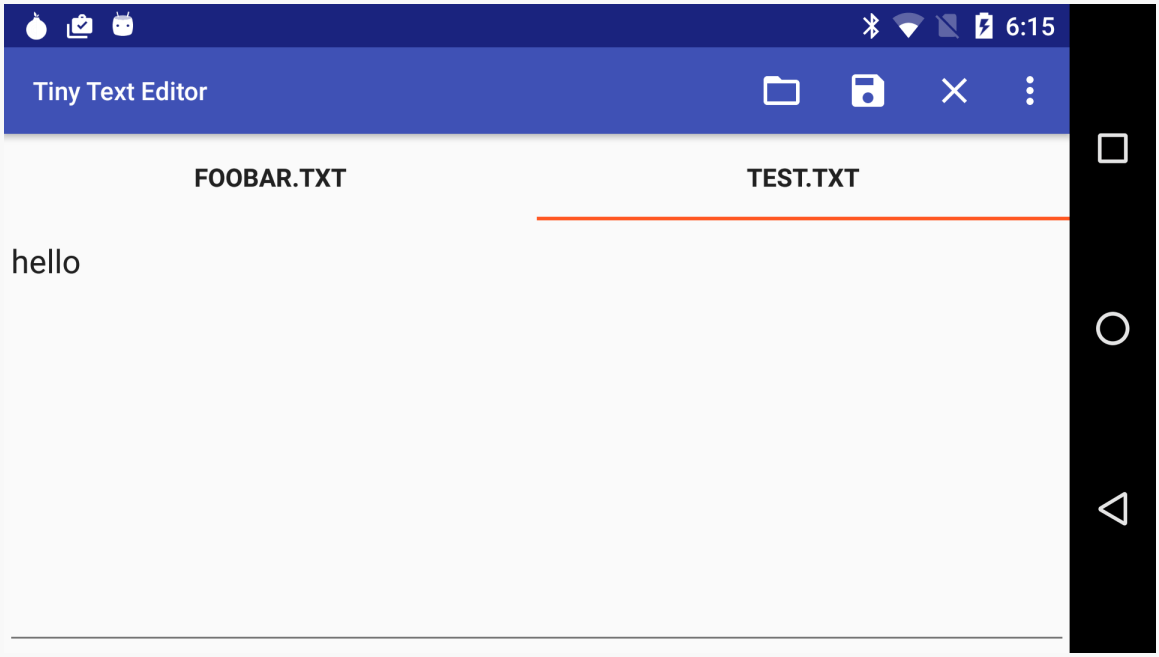


Figure 785: TinyTextEditor, Showing Two Opened Documents

We will explore the `REQUEST_CREATE` portion of `onActivityResult()` in the next section.

Creating a New Document

Given all the above, adding support for creating a new text document is fairly easy.

There is a "Create" action bar item that, when tapped, triggers a `createDocument()` call on our activity:

```
private void createDocument() {
  Intent intent=
    new Intent(Intent.ACTION_CREATE_DOCUMENT)
      .addCategory(Intent.CATEGORY_OPENABLE)
      .setType("text/plain");

  startActivityForResult(intent, REQUEST_CREATE);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

This invokes an `ACTION_CREATE_DOCUMENT` activity, asking to create a plain text file.

The user is then presented with a Storage Access Framework activity, where the user can choose a folder and fill in a filename:

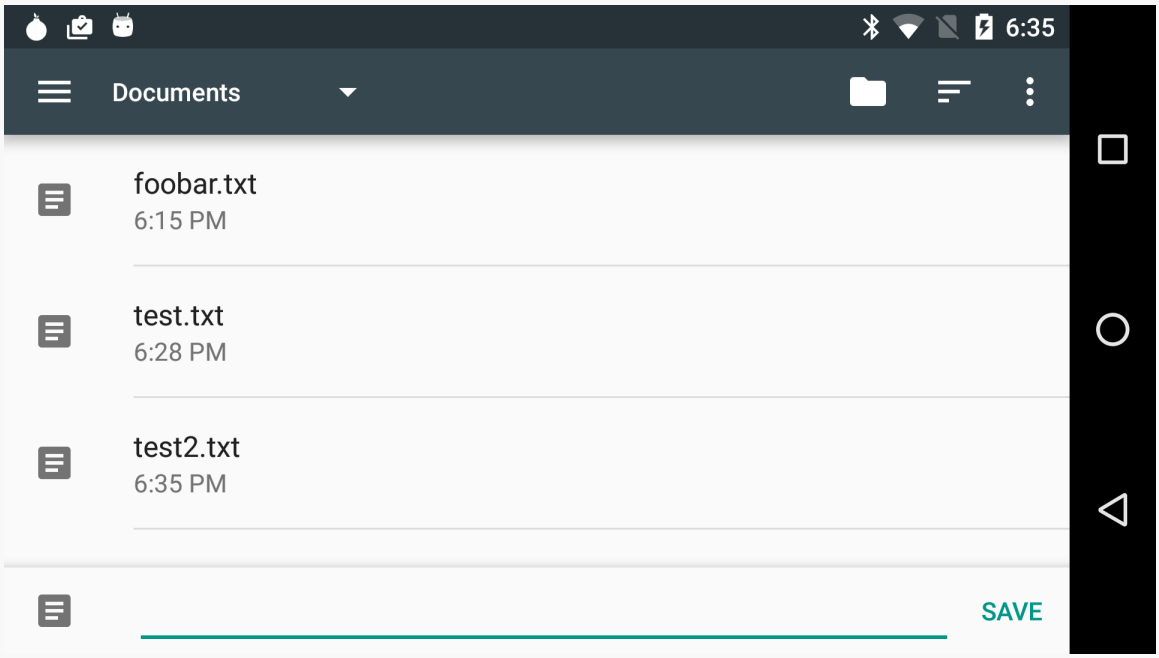


Figure 786: Storage Access Framework Create-Document Activity

If the user proceeds to fill in a filename and click "Save", `onActivityResult()` will give us a `Uri` to the empty document, which we can open in an editor:

```
    case REQUEST_CREATE:
      if (resultCode==Activity.RESULT_OK) {
        openEditor(data.getData());
      }
      break;
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

Saving a Document

This app uses a typical "low friction" approach to saving changes. While there is a "Save" action bar item to allow the user to manually save, the text is automatically saved when either the `EditorFragment` is stopped (e.g., on a configuration change) or if the user closes the document via a "Close" action bar item. The idea is that while the user *can* manually save changes, the user does not *have to* manually save changes. If you use Android Studio as an IDE, you get the same effect: while there is

a “Save” toolbar icon, it is mostly “for show”, as the IDE automatically saves changes to the files that you are editing, without manual intervention.

Both `onStop()` and `onOptionsItemSelected()` of `EditorFragment` will call a `save()` method on the fragment, though `onOptionsItemSelected()` will only do so if the user tapped the “Save” action bar item. In `save()`, we ask the `DocumentStorageService` to save the document, assuming that the document is loaded and we have write access for it:

```
private void save() {
  if (isLoaded && canWrite) {
    editor.setEnabled(false);
    DocumentStorageService.saveDocument(getActivity(),
      getDocumentUri(), editor.getText().toString(),
      isClosing);
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

While the document is being saved, we disable the editor, so the user cannot change information on the fly. Most likely that is not needed here, as we get a snapshot of the text via `editor.getText().toString()` and pass that to the service.

The `isClosing` flag passed to `saveDocument()` on `DocumentStorageService` will normally be `false` — we will see when it is `true` in the next section.

As with `loadDocument()`, `saveDocument()` handles packaging an `Intent` to invoke the `DocumentStorageService` for us:

```
public static void saveDocument(Context ctxt, Uri document,
                                String text, boolean isClosing) {
  Intent i=new Intent(ctxt, DocumentStorageService.class)
    .setAction(Intent.ACTION_EDIT)
    .setData(document)
    .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION|
              Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
    .putExtra(Intent.EXTRA_TEXT, text)
    .putExtra(EXTRA_CLOSING, isClosing);

  ctxt.startService(i);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/DocumentStorageService.java)

Eventually, that triggers a call to `save()` on the service:

```
private void save(Uri document, String text, boolean isClosing) {
  boolean isContent=
    ContentResolver.SCHEME_CONTENT.equals(document.getScheme());

  try {
    OutputStream os=
      getContentResolver().openOutputStream(document, "w");
    OutputStreamWriter osw=new OutputStreamWriter(os);

    try {
      osw.write(text);
      osw.flush();

      if (isClosing && isContent) {
        int perms=Intent.FLAG_GRANT_READ_URI_PERMISSION
          | Intent.FLAG_GRANT_WRITE_URI_PERMISSION;

        getContentResolver()
          .releasePersistableUriPermission(document, perms);
      }

      EventBus
        .getDefault()
        .post(new DocumentSavedEvent(document));
    }
    finally {
      osw.close();
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
      "Exception saving "+document.toString(), e);
    EventBus
      .getDefault()
      .post(new DocumentSaveErrorEvent(document, e));
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/DocumentStorageService.java)

Mostly, that just writes our text out to an `OutputStream` for the document `Uri`, obtained from a `ContentResolver`. We raise either a `DocumentSavedEvent` or a `DocumentSaveErrorEvent` depending on the success or failure of the operation. We will explore the `isClosing` code path in the next section.

Closing a Document

Visually, closing a document via the "Close" action bar item (looks like an X) closes the document's tab. Under the covers, we also want to save any lingering changes. Plus, we no longer need our persistable `Uri` permissions, and so it is good form to release those. And, we no longer want this document in our edit history, so if we open the app again in the future, we should not re-open this particular document.

The "Close" action bar item is defined by the activity, leading to a `closeCurrentDocument()` method:

```
private void closeCurrentDocument() {
  EditorFragment frag=adapter.getCurrentFragment();

  frag.markAsClosing();
  closeDocument(frag.getDocumentUri());
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

Here, we get our current `EditorFragment` from the `EditorsAdapter`, call a `markAsClosing()` method on it (which just sets the `isClosing` flag to `true`), then call `closeDocument()` with the `Uri` for the document:

```
private void closeDocument(Uri document) {
  if (!editHistory.removeOpenEditor(document)) {
    Toast
      .makeText(this, R.string.msg_save_history,
        Toast.LENGTH_LONG)
      .show();
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

`closeDocument()` tells the `EditorsAdapter` to remove this page, plus updates the edit history (as will be seen in detail in the next section).

Removing the `EditorFragment` from the `EditorsAdapter` has the effect of destroying that fragment. That triggers an `onStop()` call, which triggers a `save()` call as before. But, this time, we have `isClosing` set to `true`.

In the service's `save()` method, having `isClosing` set to `true` triggers the code that calls `releasePersistableUriPermission()`, where we relinquish our persistable `Uri` permission for the `Uri` for the document. However, we only try that if the `Uri` has a

content scheme — we do not need to do anything to release permissions for a file, and releasePersistableUriPermission() crashes if you try using it with a file: Uri.

Managing Edit History

The EditHistory class is responsible for persisting our roster of open documents, so that we can re-open them when the app is opened again later.

EditHistory is a singleton, accessed via the INSTANCE static field:

```
public class EditHistory {
  public static final EditHistory INSTANCE=new EditHistory();
  private static final String PREF_OPEN_EDITORS="open";
  private AtomicReference<SharedPreferences> prefsRef=
    new AtomicReference<SharedPreferences>();
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditHistory.java)

It holds onto a SharedPreferences that is being used for storage. However, since disk I/O is needed to load those SharedPreferences – which, in turn, requires a background thread — EditHistory is using an AtomicReference to ensure that we do not wind up in a state where one thread is trying to read a SharedPreferences field while another is trying to write it. This may be a bit superfluous — volatile may suffice here.

MainActivity, in onStart() calls initialize() on the EditHistory instance:

```
  public boolean initialize(Context ctxt) {
    if (prefsRef.get()==null) {
      new LoadThread(ctxt).start();

      return(false);
    }

    return(true);
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditHistory.java)

Here, if we do not already have the SharedPreferences, we kick off a thread to go load them, and we return a boolean indicating whether or not the SharedPreferences are loaded.

The LoadThread just updates the prefsRef field and raises an InitializedEvent once that work is done:

```
private class LoadThread extends Thread {
  private final Context ctxt;

  LoadThread(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
  }

  @Override
  public void run() {
    prefsRef.set(PreferenceManager.getDefaultSharedPreferences(ctxt));
    EventBus.getDefault().post(new InitializedEvent());
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditHistory.java)

MainActivity will then call its loadEditors() method at one of two times:

- In onStart(), if initialize() returned true
- In onEditHistoryInitialized(), responding to the InitializedEvent, if initialize() returned false:

```
@Subscribe(threadMode=ThreadMode.MAIN)
public void onEditHistoryInitialized(EditHistory.InitializedEvent event) {
  loadEditors();
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

loadEditors(), in turn, calls getOpenEditors() on the EditHistory. Rather than use getStringSet(), we instead use getString() on the SharedPreferences and decode it as a JSON array, using JsonReader, an Android SDK class that is to JSON as XmlPullParser is to XML:

```
public List<Uri> getOpenEditors() {
  String editors=prefsRef.get().getString(PREF_OPEN_EDITORS, null);
  ArrayList<Uri> result=new ArrayList<Uri>();

  if (editors!=null) {
    StringReader sr=new StringReader(editors);
    JsonReader json=new JsonReader(sr);

    try {
```

```
      json.beginArray();

      while (json.hasNext()) {
        result.add(Uri.parse(json.nextString()));
      }

      json.endArray();
      json.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
        "Exception reading JSON", e);
    }
  }

  return(result);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditHistory.java)

With `JsonReader`, we are expecting the JSON to be an array of strings, so we read in those strings, parse them into `Uri` objects, and use them in the `result`.

We then have `addOpenEditor()` and `removeOpenEditor()` methods that manipulate that `ArrayList`:

```
public boolean addOpenEditor(Uri document) {
  List<Uri> current=getOpenEditors();

  if (!current.contains(document)) {
    if (current.size()>9) {
      current.remove(0);
    }

    current.add(document);

    return(saveHistory(current));
  }

  return(true);
}

public boolean removeOpenEditor(Uri document) {
  List<Uri> current=getOpenEditors();

  current.remove(document);
```

```
    return(saveHistory(current));
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditHistory.java)

`addOpenEditor()` caps the `ArrayList` length to 9, so we do not open some ridiculous number of tabs when the app is next launched.

Both `addOpenEditor()` and `removeOpenEditor()` call a `saveHistory()` method to persist the revised `ArrayList`:

```
  private boolean saveHistory(List<Uri> history) {
    StringWriter sw=new StringWriter();
    JsonWriter json=new JsonWriter(sw);

    try {
      json.beginArray();

      for (Uri uri : history) {
        json.value(uri.toString());
      }

      json.endArray();
      json.close();

      prefsRef
        .get()
        .edit()
        .putString(PREF_OPEN_EDITORS, sw.toString())
        .apply();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
        "Exception saving JSON", e);

      return(false);
    }

    return(true);
  }
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditHistory.java)

`saveHistory()` reverses the work of `getOpenEditors()`: we encode the `ArrayList` into a JSON array of strings, then save that in the `SharedPreferences`.

Managing Multiple Windows

The `EditorFragment` adds a "Launch" option to the action bar (looks like a square with an arrow). When tapped, if the device is in multi-window mode, we want to open this document in a separate window, removing its tab from the current activity. However, "Launch" is only shown if we are running on Android 7.0 *and* are in multi-window mode:

```
@Override
public void onCreateOptionsMenu(Menu menu,
                                MenuInflater inflater) {
  inflater.inflate(R.menu.editor_actions, menu);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.N) {
    launchItem=menu.findItem(R.id.launch);
    launchItem.setVisible(getActivity().isInMultiWindowMode());
  }

  super.onCreateOptionsMenu(menu, inflater);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

The "Launch" option eventually triggers a `launchInNewWindow()` method on the `MainActivity`:

```
@Override
public void launchInNewWindow(Uri document) {
  adapter.remove(document);

  Intent i=
    new Intent(this, MainActivity.class)
      .setAction(ACTION_NEW_WINDOW)
      .setData(document)
      .setFlags(Intent.FLAG_ACTIVITY_LAUNCH_ADJACENT |
        Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_MULTIPLE_TASK);

  startActivity(i);
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/MainActivity.java)

Here, we remove our tab and request to open a fresh copy of our `MainActivity`, with the `Uri` of the document put into the `Intent`, with our custom `ACTION_NEW_WINDOW` action, and with `FLAG_ACTIVITY_LAUNCH_ADJACENT` to get a new window. `onCreate()`

of the new `MainActivity` instance will see that it was passed a `Uri` with `ACTION_NEW_WINDOW` and will open that document, instead of trying to open the edit history.

Since the user might toggle multi-window mode while our activity already exists, `EditorFragment` overrides `onMultiWindowModeChanged()` and updates the visibility of the "Launch" action bar item to match:

```
@Override
public void onMultiWindowModeChanged(boolean inMultiWindow) {
  super.onMultiWindowModeChanged(inMultiWindow);

  if (launchItem!=null) {
    launchItem.setVisible(inMultiWindow);
  }
}
```

(from Documents/TinyTextEditor/app/src/main/java/com/commonsware/android/tte/EditorFragment.java)

Here, we implement the one-parameter version of `onMultiWindowModeChanged()`, to be compliant with Android 7.0 and 7.1, despite the fact that it was deprecated in Android 8.0.

ACTION_EDIT and Tasks

We have logic in the app already to handle `ACTION_EDIT`: if the `Intent` used to create our activity has a `Uri`, we open it.

However, what happens if our app is already open, and the user triggers an `ACTION_EDIT` request from another app?

By default, what would happen is that we would open another instance of our activity, separate from the original one. That is not a good user experience for this sort of app, particularly in a single-window environment. Instead, we should pick up the `Intent` in our existing activity instance and just open a new tab.

To implement that, ideally we would add `android:launchMode="singleTop"` to our `<activity>` element in the manifest, along with our `<intent-filter>` for `ACTION_EDIT`. As is covered in the chapter on tasks, `singleTop` will deliver our `Intent` to an existing activity instance... if that instance is on the top of our BACK stack. Unfortunately, this breaks the current multi-window support in Android 7.0.

What's Missing

This app is *far* from perfect. After all, it is a tiny text editor, not a full-featured, all-the-bells-and-whistles, hey-let's-go-find-investors sort of text editor.

Of note:

- We always save the text to disk when `save()` gets called... even if there have been no changes to the content
- We wind up making copies of the text a lot, which can be a problem with larger documents due to heap fragmentation
- On a configuration change, we save and reload the text contents, rather than optimizing for that scenario and trying to hold onto the text
- It is possible that there will be some order-of-operations issues when a new window is launched, depending on the timing of `onStop()` of the original `EditorFragment` and when the new fragment in the new activity instance is created

Document Trees

`ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT` are sufficient for most apps. These roughly correspond to the "file open" and "new file" dialogs that you see in desktop operating systems.

However, there may be cases where you need the equivalent of a "choose folder" dialog, to allow the user to pick a location where you can create (or work with) several documents. For example, suppose that your app offers a report generator, taking data from the database and creating a report with tables and graphs and stuff. Some file formats, like PDF, might have the entire report in a single file — for that, use `ACTION_CREATE_DOCUMENT` to allow the user to choose where to put that report. Other file formats, like HTML, might require several files (e.g., the report body in HTML and embedded graphs in PNG format). For that, you really need a "folder", into which you can create all of those individual bits of content.

For that, the Storage Access Framework offers document trees... as of Android 5.0 (API Level 21). Android 4.4's edition of the Storage Access Framework lacked this capability.

Getting a Tree

Instead of using ACTION_OPEN_DOCUMENT, you can use ACTION_OPEN_DOCUMENT_TREE. Once again, you will use startActivityForResult() to request access to the tree. In onActivityResult(), the result Intent has a Uri (getData()) that represents the tree. You should have full read/write access not only to this tree but to anything inside of it.

Another option, starting with Android 7.0, is "scoped directory access". Here, you work with StorageManager to access the device's StorageVolume list. All devices should have at least one StorageVolume, representing what we think of as external storage. Some devices may have more than that, representing mounted removable media. Given a StorageVolume, you can call createAccessIntent() to get an Intent that will ask the user permission for access to some portion of that volume, when called with startActivityForResult(). As with ACTION_OPEN_DOCUMENT_TREE, you get a Uri in onActvityResult() that you can then use to work with that tree of files.

Working in the Tree

The simplest approach for then working with the tree is to use the aforementioned DocumentFile wrapper. You can create one representing the tree by using the fromTreeUri() static method, passing in the Uri that you got from the ACTION_OPEN_DOCUMENT_TREE request.

From there, you can:

- Call listFiles() to get the immediate children of the root of this tree, getting back an array of DocumentFile objects representing those children
- Call isDirectory() to confirm that you do indeed have a tree (or, call it on a child to see if that child represents a sub-tree)
- For those existing children that are files (isFile() returns true), use getUri() to get the Uri for this child, so you can read its contents using a ContentResolver and openInputStream()
- Call createDirectory() or createFile() to add new content as an immediate child of this tree, getting a DocumentFile as a result
- For the createFile() scenario, call getUri() on the DocumentFile to get a Uri that you can use for writing out the content using ContentResolver and openOutputStream()
- and so on

Note that you can call `takePersistableUriPermission()` on a `ContentResolver` to try to have durable access to the document tree, just as you can for a `Uri` to an individual document.

Getting a Tree: Example

The `Documents/DocumentTree` sample application demonstrates how to use `ACTION_OPEN_DOCUMENT_TREE` and `StorageManager`/`StorageVolume` to get a `Uri` pointing to a directory that you can work with.

The Objective: a Preference for Storage

The sample app's UI is a `PreferenceFragment`, where we have two preferences: one to pick a document tree via `ACTION_OPEN_DOCUMENT_TREE` and one to pick a `StorageVolume` from among the available volumes. In theory, an app might include one of these for the user to pick an alternative default storage location for files, for example. However, since the `StorageVolume` APIs for choosing a storage volume are new to API Level 24, we will only enable that preference on compatible devices.

In each case, part of the work to get access to these locations involves `startActivityForResult()`, which is unusual for a preference and adds to the sample's complexity.

What the User Sees

When the user first launches the app, the preference subtitles are "no value", because the user has not chosen anything yet:

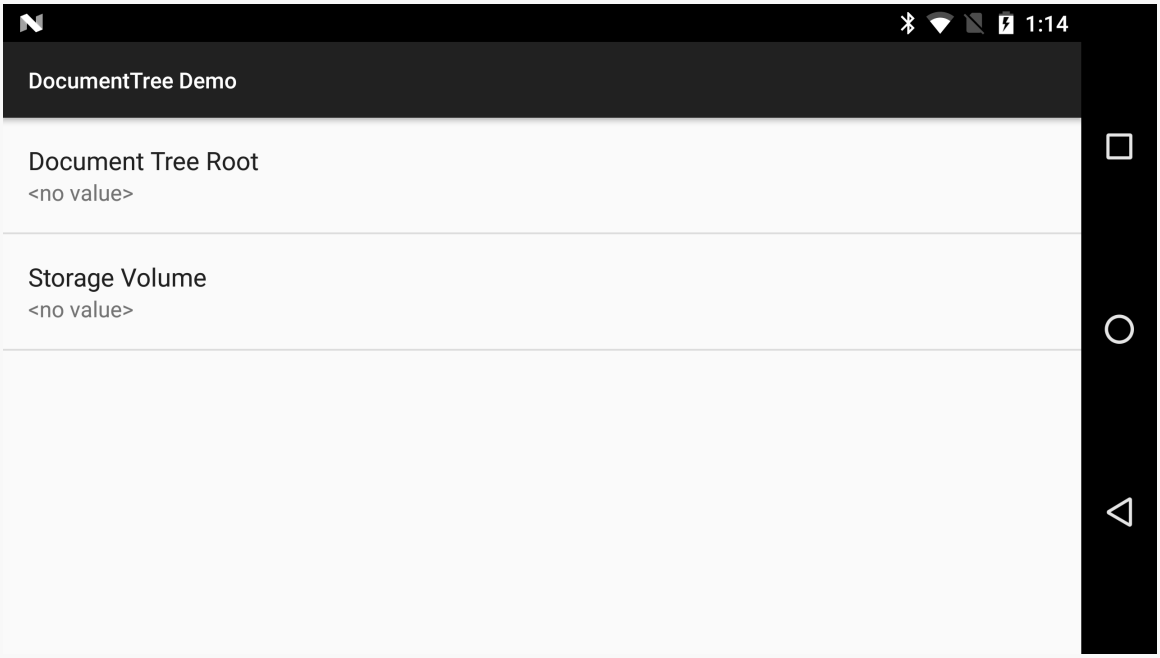


Figure 787: DocumentTree Demo, As Initially Launched

If the user taps the “Document Tree Root” preference, the UI for the Storage Access Framework appears, allowing the user to browse for a directory of interest:

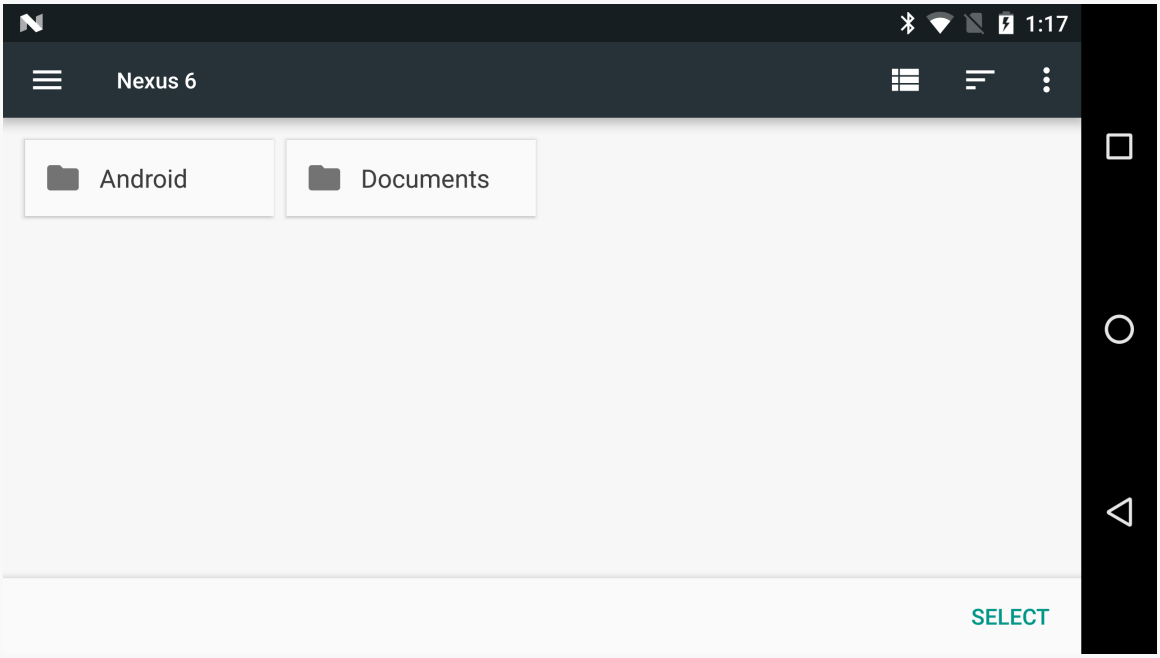


Figure 788: DocumentTree Demo, Showing "Internal Storage" via SAF

If the user chooses a location, the preference is updated with the `Uri` of the selected document tree:

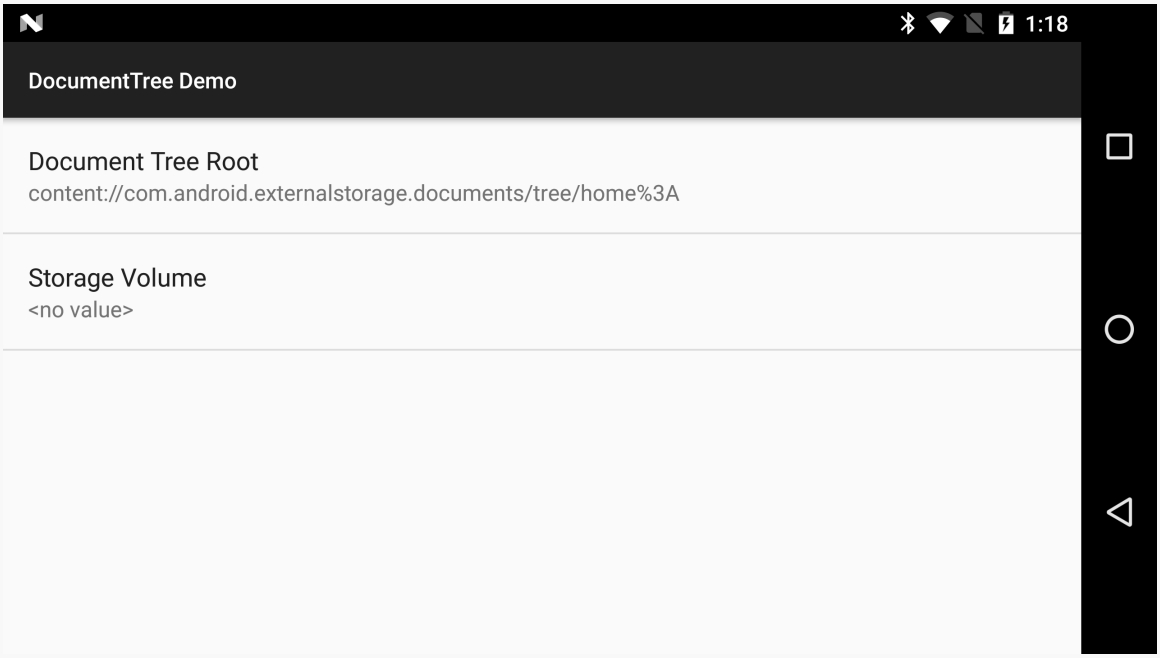


Figure 789: DocumentTree Demo, Showing Selected Document Tree Uri

If the user taps the "Storage Volume" preference, a `ListPreference` dialog appears, showing the available storage volumes:

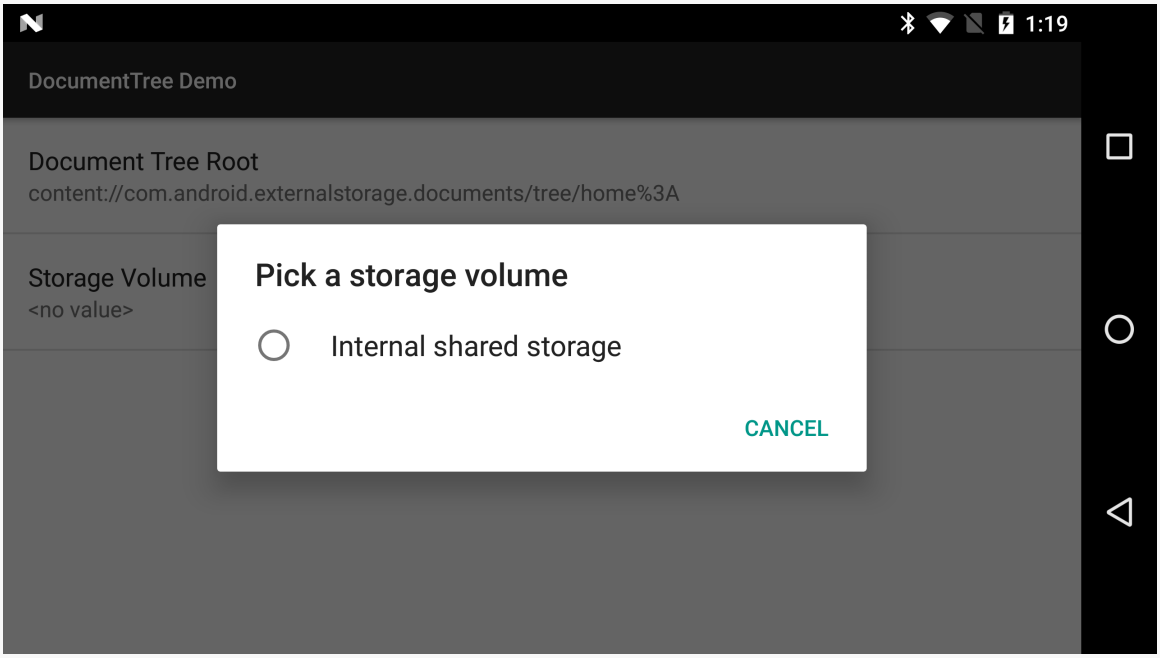


Figure 790: DocumentTree Demo, Showing Available Storage Volume(s)

On some devices, there will only be one option (“Internal shared storage”, or what we developer call “external storage”). On other devices, if there is a piece of removable storage mounted, there will be more than one option.

If the user chooses a volume, a permission confirmation dialog may appear, to confirm that the user wants to grant you access to the “Documents” directory inside of that storage volume:

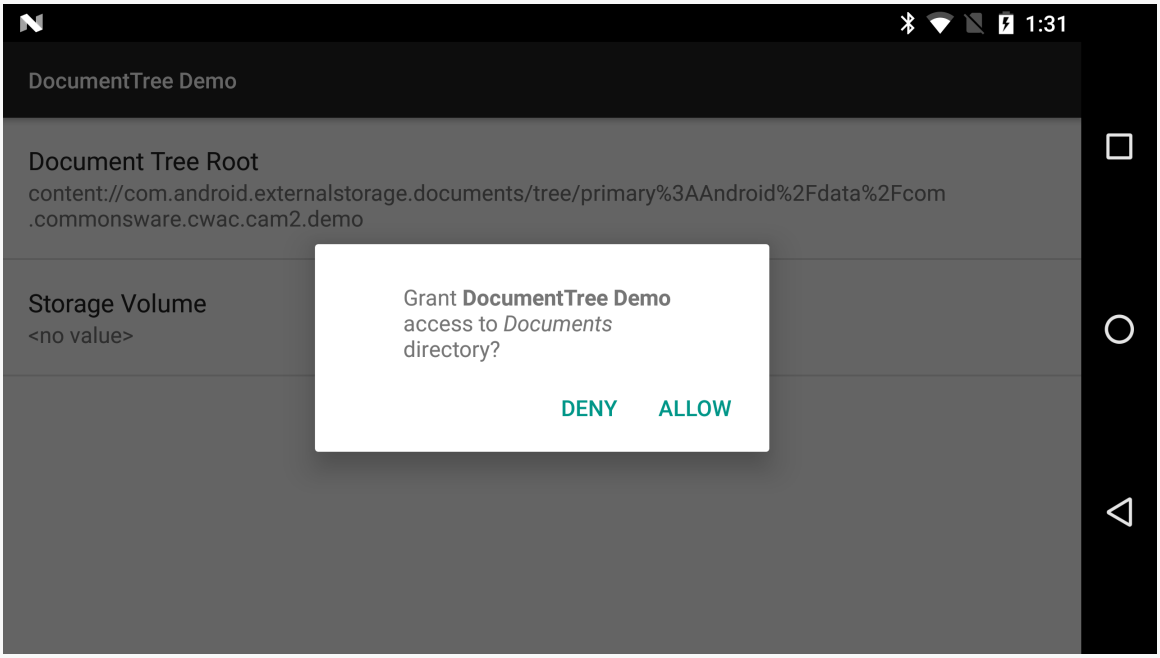


Figure 791: DocumentTree Demo, Requesting Permission

If the user grants permission, once again the preference's subtitle will reflect the `Uri` of the chosen location:

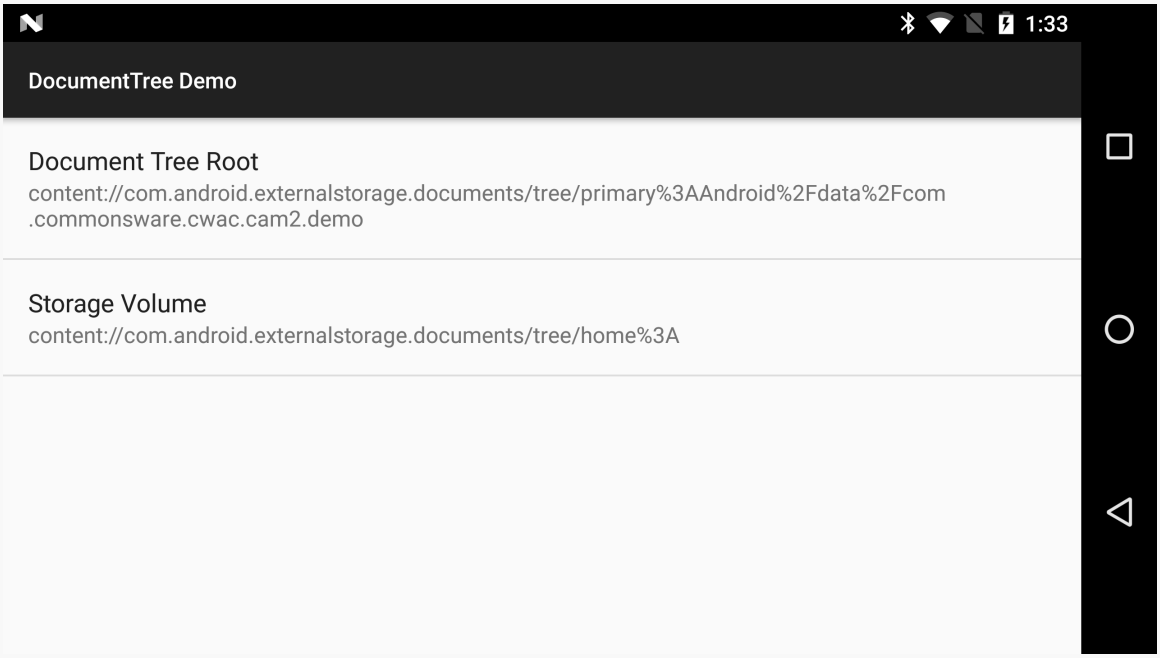


Figure 792: DocumentTree Demo, Showing Selected Storage Volume Directory

On Android 5.0-6.0 devices, the app will run, but the storage volume preference is disabled:

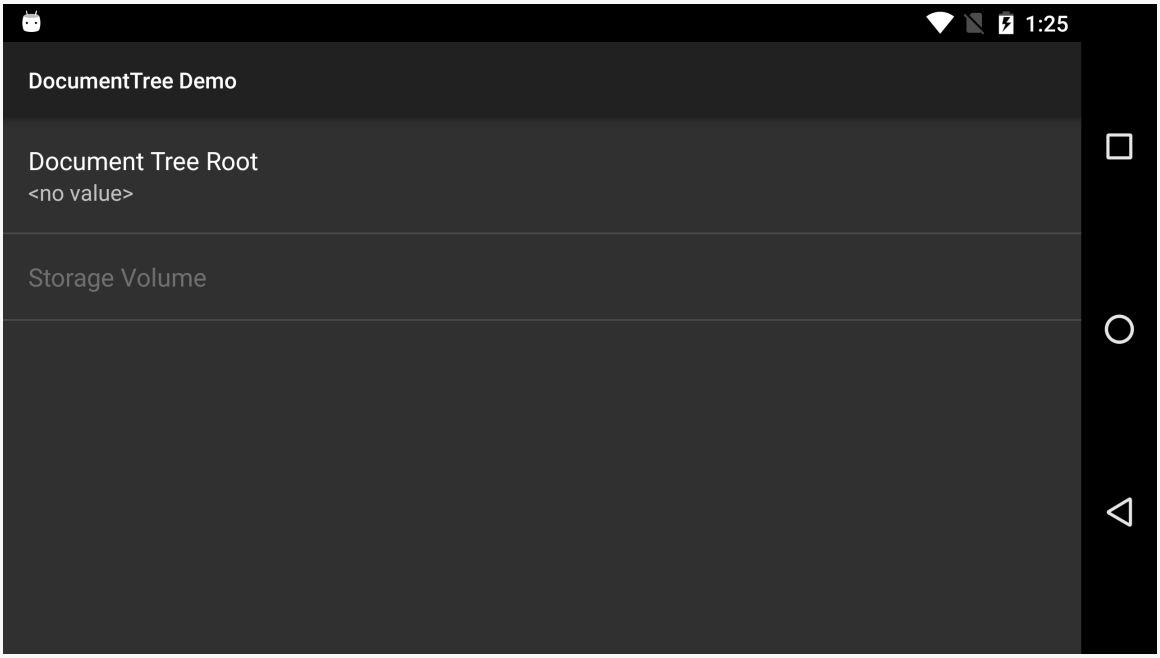


Figure 793: DocumentTree Demo, Running on Android 6.0

The Document Tree

Of the two options, `ACTION_OPEN_DOCUMENT_TREE` is the most straight-forward to implement: call `startActivityForResult()` and get your `Uri` in `onActivityResult()`.

But, since preferences are not set up to handle `startActivityForResult()` or receive data via `onActivityResult()`, we have a little bit of work to do.

The Preference XML

The app has a `res/raw/settings.xml` file containing our preferences:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
  <Preference
    android:key="documentTree"
    android:title="@string/pref_doc_tree" />
  <ListPreference
    android:dialogTitle="@string/dlg_storage_volume"
    android:enabled="@bool/is_nougat"
    android:key="storageVolume"
```

```
    android:title="@string/pref_storage_volume" />
</PreferenceScreen>
```

(from Documents/DocumentTree/app/src/main/res/xml/settings.xml)

The first one is our "Document Tree Root" preference... and it is literally a `Preference`. This is not used all that frequently, since it cannot actually collect any preference data. In cases like this one, where we really want to handle this more like the user tapped on a generic `ListView` row, it is a reasonable choice.

We will explore the `ListPreference` for the "Storage Volume" option later in this section.

Populating the Preference

The UI is a `PreferenceFragment` subclass named `SettingsFragment`. In `onCreate()`, we call `addPreferencesFromResource()` to inflate that preference XML and populate the fragment:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  addPreferencesFromResource(R.xml.settings);

  prefDocTree=findPreference(PREF_DOC_TREE);
  prefs=prefDocTree
    .getSharedPreferences();
  prefs.registerOnSharedPreferenceChangeListener(this);
  onSharedPreferenceChanged(prefs, PREF_DOC_TREE);
  docTreeHelper=new DocumentHelper(this,
    prefDocTree);

  prefVolumes=(ListPreference)findPreference(PREF_VOLUMES);

  if (prefVolumes.isEnabled()) {
    populateVolumes();
    onSharedPreferenceChanged(prefs, PREF_STORAGE_URI);
    volumeHelper=
      new VolumeHelper(this, prefVolumes,
        PREF_STORAGE_URI, Environment.DIRECTORY_DOCUMENTS);
  }
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/SettingsFragment.java)

We then do a few things to set up our "Document Tree Root" preference:

- We call `findPreference()` to get that `Preference` object, storing it in a `prefDocTree` field.
- We ask the `Preference` for the `SharedPreferences` that are being used, holding onto that in a field named `prefs`.
- Register the fragment itself as an `OnSharedPreferenceChangeListener` for the `SharedPreferences`, then immediately call `onSharedPreferenceChanged()`. That, in turn, fills in the summary of the `Preference` with the current `Uri`, if we have one:

```
@Override
public void onSharedPreferenceChanged(SharedPreferences prefs,
                                      String key) {
  if (PREF_DOC_TREE.equals(key)) {
    prefDocTree.setSummary(prefs.getString(key, "<no value>"));
  }
  else if (PREF_STORAGE_URI.equals(key)) {
    prefVolumes
      .setSummary(prefs
        .getString(key, "<no value>").replaceAll("%", "%%"));
  }
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/SettingsFragment.java)

- Wrap that `Preference` in a `DocumentHelper` object, which we will look at shortly.

We will cover the remainder of this code, pertaining to the other preference, later.

Choosing a Tree

We need some common code between the document-root and the storage-volume options:

- Bridging between a `Preference` and a hosting activity or fragment that can do the `startActivityForResult()` and `onActivityResult()` work
- Calling `takePersistableUriPermission()`
- Updating the `SharedPreferences` with the `Uri` that we receive

The `TreeUriPreferenceHelper` abstract class, along with its `DocumentHelper` and `VolumeHelper` subclasses, implement this common code.

A `TreeUriPreferenceHelper` subclass' constructor needs to be passed the `Preference` that we are "helping", along with some implementation of the `Host` interface:

```
  public interface Host {
    void startActivityForHelper(Intent intent,
                                TreeUriPreferenceHelper helper);
  }
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/TreeUriPreferenceHelper.java)

`DocumentHelper` simply collects those values, passes them to `TreeUriPreferenceHelper`, and registers itself to be called when the user clicks on the `Preference`:

```
package com.commonsware.android.documenttree;

import android.content.Intent;
import android.preference.Preference;

public class DocumentHelper extends TreeUriPreferenceHelper
  implements Preference.OnPreferenceClickListener {
  public DocumentHelper(Host host, Preference pref) {
    super(host, pref);
    pref.setOnPreferenceClickListener(this);
  }

  @Override
  protected String getUriKey() {
    return (pref.getKey());
  }

  @Override
  public boolean onPreferenceClick(Preference preference) {
    Intent i=new Intent(Intent.ACTION_OPEN_DOCUMENT_TREE);

    host.startActivityForHelper(i, this);

    return(true);
  }
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/DocumentHelper.java)

`TreeUriPreferenceHelper`, in turn, just holds onto the `Host` and `Preference` in `host` and `pref` fields, respectively.

When the user clicks on the `Preference`, the `onPreferenceClick()` method of the `DocumentHelper` is called. There, we create an `ACTION_OPEN_DOCUMENT_TREE` `Intent` and call `startActivityForHelper()` on the `host`.

Our `Host`, in this case, is the `SettingsFragment`, so it has an implementation of `startActivityForHelper()`:

```
@Override
public void startActivityForHelper(Intent intent,
                                   TreeUriPreferenceHelper helper) {
  if (helper==docTreeHelper) {
    startActivityForResult(intent, REQUEST_DOC_TREE);
  }
  else if (helper==volumeHelper) {
    startActivityForResult(intent, REQUEST_STORAGE_VOLUME);
  }
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/SettingsFragment.java)

It just sees which `TreeUriPreferenceHelper` we are working with, then calls `startActivityForResult()` with an appropriate request code (e.g., `REQUEST_DOC_TREE`).

Eventually, `SettingsFragment` should be called with `onActivityResult()`. If the result is `RESULT_OK`, we forward the result along to the `TreeUriPreferenceHelper`, based on the request code:

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                             Intent data) {
  if (resultCode==Activity.RESULT_OK) {
    if (requestCode==REQUEST_DOC_TREE) {
      docTreeHelper.onActivityResult(data);
    }
    else if (requestCode==REQUEST_STORAGE_VOLUME) {
      volumeHelper.onActivityResult(data);
    }
  }
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/SettingsFragment.java)

`TreeUriPreferenceHelper` has the common implementation of `onActivityResult()`, where we call `takePersistableUriPermission()` (asking for read/write access) and put the `Uri` into the `SharedPreferences` under some key:

```
public void onActivityResult(Intent data) {
  Uri docTree=data.getData();
  ContentResolver cr=pref.getContext().getContentResolver();
  int perms=Intent.FLAG_GRANT_READ_URI_PERMISSION
    | Intent.FLAG_GRANT_WRITE_URI_PERMISSION;

  cr.takePersistableUriPermission(docTree, perms);

  pref
    .getSharedPreferences()
    .edit()
    .putString(getUriKey(), docTree.toString())
    .apply();
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/TreeUriPreferenceHelper.java)

In the case of the document-root `Preference`, that key is the key from the `Preference` itself (`getKey()`). Saving the value not only persists it, but it also triggers the `SettingsFragment` to be notified about the new value, causing `SettingsFragment` to update the `Preference` summary... which is why we see the `Uri` show up on the screen right after selecting it.

The Storage Volume

The `StorageVolume` scenario is a bit more complicated, in that we have to provide the UI for choosing a volume — this is not provided by Android. That, plus some interesting challenges in the `StorageVolume` implementation, add to our level of effort.

The Preference XML

The `settings.xml` file has a `ListPreference` that will serve as the UI for selecting a `StorageVolume`:

```
<ListPreference
  android:dialogTitle="@string/dlg_storage_volume"
  android:enabled="@bool/is_nougat"
  android:key="storageVolume"
```

(from Documents/DocumentTree/app/src/main/res/xml/settings.xml)

Since the roster of possible volumes is dynamic, we cannot provide our `ListPreference` contents via `string-array` resources, but instead will need to do so from Java code.

Note that this preference is enabled based upon an `is_nougat` `bool` resource. That is set to `true` for API Level 24+ (in `res/values-v24/bools.xml`), `false` otherwise (in `res/values/bools.xml`).

Populating the Preference

Some of the work from `onCreate()` of `SettingsFragment` is for setting up this `ListPreference`:

```
    prefVolumes=(ListPreference)findPreference(PREF_VOLUMES);

    if (prefVolumes.isEnabled()) {
      populateVolumes();
      onSharedPreferenceChanged(prefs, PREF_STORAGE_URI);
      volumeHelper=
        new VolumeHelper(this, prefVolumes,
          PREF_STORAGE_URI, Environment.DIRECTORY_DOCUMENTS);
    }
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/SettingsFragment.java)

We store the `ListPreference` in a `prefVolumes` field, before calling a private `populateVolumes()` method to fill in the list contents. We also trigger updating its summary via a manual call to `onSharedPreferenceChanged()`, plus wrap the `ListPreference` in a `VolumeHelper` that we will explore in detail shortly. And all of this is wrapped in a check to see if the preference is enabled; if it is not, we skip this work, as it is unnecessary.

`populateVolumes()` is responsible for providing the entries and values for the `ListPreference`, based on the available volumes:

```
@TargetApi(Build.VERSION_CODES.N)
private void populateVolumes() {
  StorageManager storage=
    (StorageManager)getActivity()
      .getSystemService(Context.STORAGE_SERVICE);
  List<StorageVolume> volumes=storage.getStorageVolumes();
```

```
    Collections.sort(volumes, new Comparator<StorageVolume>() {
      @Override
      public int compare(StorageVolume lhs,
                         StorageVolume rhs) {
        return(lhs.getDescription(getActivity())
          .compareTo(rhs.getDescription(getActivity())));
      }
    });

    String[] displayNames=new String[volumes.size()];
    String[] uuids=new String[volumes.size()];

    for (int i=0;i<volumes.size();i++) {
      displayNames[i]=volumes.get(i).getDescription(getActivity());
      uuids[i]=volumes.get(i).getUuid();

      if (uuids[i]==null) {
        uuids[i]=STORAGE_FAKE_UUID;
      }
    }

    prefVolumes.setEntries(displayNames);
    prefVolumes.setEntryValues(uuids);
  }
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/SettingsFragment.java)

We start off by getting a `StorageManager` system service. Here, we are using the newer version of `getSystemService()`, introduced in API Level 21, where we can pass in the Java class of the system service that we want (`StorageManager.class`). This allows Android to return an instance of the actual class, avoiding a cast.

Then, we call `getStorageVolumes()` on the `StorageManager`, to get the roster of available `StorageVolume` objects.

Since those `StorageVolume` objects might arrive in any order, we `sort()` them by their description, which is a human-readable label describing what the volume is. For example, for removable storage, it might be a combination of the manufacturer of the drive or card, plus the stated capacity of the drive or card.

Since `ListPreference` wants two `String` arrays for the entries and values, we set those up, filling them in from the description and UUIDs of the volumes. Each volume is supposed to have a UUID, but that is not guaranteed — in particular, the `StorageVolume` for external storage returns `null` for `getUuid()`. Since `ListPreference` *really* does not like `null` values, we substitute in a non-UUID string

(`STORAGE_FAKE_UUID`, defined to be `"fake"`) to identify it. We then give those two string arrays to the `ListPreference`.

Choosing a Volume

`VolumeHelper`, like `DocumentHelper`, is designed to help bridge between the preference system, the hosting fragment, and the Android APIs for getting a document tree `Uri`.

`VolumeHelper` takes two additional constructor parameters, beyond the `Host` and `Preference`:

- The key for the `SharedPreference` under which the `Uri` will be stored. The `ListPreference` will store the UUID of the storage volume under its key, but once we get the `Uri`, we need to save it to the `SharedPreferences` as well, for later use.
- The directory on the storage volume to use. `createAccessIntent()` accepts any of the standard `Environment` directories — in this case, we are using `DIRECTORY_DOWNLOADS`

The constructor holds onto those additional parameters in fields, then registers itself to respond to when the `ListPreference` value changes, because the user selected a different `StorageVolume` in the list:

```
package com.commonsware.android.documenttree;

import android.content.Intent;
import android.os.storage.StorageManager;
import android.os.storage.StorageVolume;
import android.preference.ListPreference;
import android.preference.Preference;
import java.util.List;

public class VolumeHelper extends TreeUriPreferenceHelper
  implements Preference.OnPreferenceChangeListener {
  private final String uriKey;
  private final String dirName;

  public VolumeHelper(Host host, ListPreference pref, String uriKey,
                      String dirName) {
    super(host, pref);

    this.uriKey=uriKey;
    this.dirName=dirName;
```

```
    pref.setOnPreferenceChangeListener(this);
  }

  @Override
  protected String getUriKey() {
    return(uriKey);
  }

  @Override
  public boolean onPreferenceChange(Preference pref,
                                    Object o) {
    StorageManager storage=
      pref.getContext().getSystemService(StorageManager.class);
    List<StorageVolume> volumes=storage.getStorageVolumes();
    String uuid=o.toString();

    for (StorageVolume volume : volumes) {
      if ((volume.getUuid()==null &&
        uuid.equals(SettingsFragment.STORAGE_FAKE_UUID)) ||
        (uuid.equals(volume.getUuid()))) {
        Intent i=volume.createAccessIntent(dirName);

        host.startActivityForHelper(i, this);
        break;
      }
    }

    return(true);
  }
}
```

(from Documents/DocumentTree/app/src/main/java/com/commonsware/android/documenttree/VolumeHelper.java)

When the user eventually does change the selection in the `ListPreference` and `onPreferenceChange()` is called, we get a fresh `StorageManager`. Unfortunately, `StorageManager` does not have any sort of lookup API to get a `StorageVolume` by UUID, so we have to iterate over the currently-available volumes and find a match, taking into account our fake UUID for the `null`-UUID case.

When we find a match, we call `createAccessIntent()` on the `StorageVolume`, passing in the directory name. That `Intent` is then given to the `Host` via `startActivityForHelper()`. That will trigger the same process as was used for `DocumentHelper`, eventually resulting in the `Uri` being saved to the `SharedPreferences` under the supplied key.

Potential Issues

It is unclear if more than one `StorageVolume` could have a `null` UUID. If it can, the approach of using a fake value in lieu of `null` will not work. Of course, if more than one `StorageVolume` could have a `null` UUID, we will have no means of identifying *which* `StorageVolume` the `null` UUID refers to, making long-term identification of `StorageVolume` objects difficult.

Scoped Directory Access Bug

The scoped directory access feature — using `StorageVolume` to request access to standard directories — works fairly well on Android 7.0+... with one UX flaw, tied to how the user grants you that access.

The flow of the permission dialogs resembles that of Android 6.0's runtime permissions:

- When you first ask for access, the user can allow or deny
- If the user denied access, and you later ask for access again, the dialog now has a "Don't ask again" checkbox
- If the user checked that checkbox and denied access again, any future attempts you make to request access will be denied immediately, without the user seeing a dialog

One problem is that we have no good way of knowing that the user has previously denied our request, let alone checked the "Don't ask again" checkbox. With Android 6.0's runtime permissions, we have `checkSelfPermission()` and `shouldShowPermissionRequestRationale()` for those things. We have no equivalents for scoped directory access.

However, the bigger problem is that once the user checks "Don't ask again" and denies access, *the user has no further recourse*. With runtime permissions, the user can always go into the Permissions area of an app's page in Settings and manually grant permissions. There is no equivalent of this for the scoped directory access API.

On Android 7.1, the user can use "Clear Data" to reset these dialogs, causing future dialogs to appear again even if "Don't ask again" had been checked. Of course, "Clear Data" has somewhat broader impact than this, and the user might not appreciate wiping out all the app's local data.

Worse, on Android 7.0, not only does "Clear Data" not fix this, but *a full uninstall of the app* does not fix this. Nothing short of a factory reset will allow the app to ask the user for permission and the user have an opportunity again to grant permission.

Admittedly, this is an edge case, but it is one that you should keep in mind if you are using `createAccessIntent()` and the scoped directory access API. Keep an eye on this issue to try to get some resolution to how the user is supposed to manually revert the "Don't ask again" status.

Android 8.0 Changes

Android 8.0 introduced some new options for consuming documents and document trees via the Storage Access Framework.

Document Tree Traversal

The `DocumentsContract` client interface to the Storage Access Framework now has a `findDocumentPath()` method. Many developers will think that this means that they can get a filesystem path for a document `Uri` from the Framework.

That is not what this method does.

Instead, `findDocumentPath()` returns a list of document IDs representing the hierarchy of document trees leading to the specific document being requested. In other words, if the document is inside of a `goo` tree, which is inside of a `bar` tree, which is inside of a `foo` tree, and `foo` is the top of a provider's hierarchy, `findDocumentPath()` would return a `List` of document IDs representing `foo`, `bar`, `goo`, and the requested document.

Presumably, the idea is that this would be used by client UIs to help build a way to traverse the relevant set of document trees for a particular document.

For this to work, the corresponding `DocumentsProvider` needs to implement its own `findDocumentPath()` method.

Web Links for Documents

`DocumentsContract` also has a `createWebLinkIntent()` method, with a corresponding implementation on `DocumentsProvider`. This is very poorly documented, but apparently the idea is that for some cloud document providers,

you can get a URL to the document, given the Storage Access Framework `Uri` for the document. Presumably, this is a publicly-visible document, and the URL could be sent to other parties (e.g., via email) for them to see the document.

Android itself does not ship with a cloud document provider. Google Play devices may ship with Google Drive, and Drive might support this feature.

Document Settings Activity

If you implement a `DocumentsProvider`, you have the option of including `FLAG_SUPPORTS_SETTINGS` in the details that you return for `queryChildDocuments()`. If you do that, and you have an activity that supports `ACTION_DOCUMENT_SETTINGS`, the user may be presented with an option to visit that activity. The activity should be given the `Uri` of the particular document that the user wishes to manage. A provider might use this for offering management of metadata for the document: tags, access rights, etc.

Providing Documents

The Storage Access Framework gives developers access to `ACTION_OPEN_DOCUMENT` and related `Intent` actions to perform operations on a document provider.

However, what if you want to *be* a document provider?

To do that, you will need to create a subclass of `DocumentsProvider`, override some abstract methods, and perhaps put up with some really obtuse error messages.

This chapter will help you in setting up your `DocumentsProvider`. With luck, you will escape without encountering errors.

Prerequisites

This chapter assumes that you have read the preceding chapter on consuming documents, along with its prerequisites.

Have Your Content, and Provide it Too

Most apps will not need to implement a document provider. They might not even consume documents, let alone provide them to other apps.

However, if your app has document-style content, and that content is of a MIME type that could reasonably be manipulated by other apps, you should consider implementing a document provider to allow the user to manipulate that content using those other apps.

Historically, developers had two main approaches for content:

1. Store it on internal storage
2. Store it on external storage

The internal storage route would be for content that would not normally be user-accessible, while external storage would be for user-accessible content.

In either of those cases, you will want to consider creating a document provider. In the case of internal storage, the user has no way to get to that content except through your app, and so if your app does not offer some capabilities that other apps do for that content, you limit your user by not having a document provider. In the case of external storage, while users in theory could use a file manager or something to try to get other apps to recognize the content, it will be easier for users if you provide a document provider to proactively publish this content to consuming apps.

However, bear in mind that your app does not have to store its content in either of these places. It could store the content in *somebody else's* document provider, using the mechanisms discussed in the preceding chapter on consuming documents. In this case, you would not need to publish a document provider yourself, as the content is available through the same provider that your code is using.

It may also be the case that while your app is the one directly storing the content on internal or external storage, that content would not reasonably be used by other apps. Perhaps it is in some non-standard format that other apps are unlikely to support. Perhaps the files are not to be used by other apps for security reasons. In these cases, skipping the document provider is reasonable, though not exactly ideal from the user's standpoint. In particular, it restricts them from using those files with apps that can work with any file, such as attaching them to email messages.

So, for example:

- If you are implementing a camera app, and you are not storing the photos in the standard `DIRECTORY_DCIM` location for photos, but you are storing the photos yourself in files consider implementing a document provider so users can get at the photos you are taking. But, if you are implementing a camera app, you might elect to allow the user to indicate some place in somebody else's document provider where you could save the photo on their behalf.
- If you are implementing some sort of network-synced file service (e.g., DropBox, Bittorrent Sync), or some sort of on-device version control system, consider implementing a document provider so users can manipulate the documents that you are managing on their behalf.

Key Provider Concepts

Creating a document provider is significantly more complex than is creating a document consumer. To help make sense of what is required, here are some key terms that you will need to understand.

Roots

A document provider can publish one or more “roots”. Basically, a “root” points to a tree of documents. Many providers will have just one root, but it is entirely possible for your provider to have more than one.

Documents

Documents, in turn, represent what in filesystem terms would be considered files and directories. A document can either have children (e.g., a directory) or it can have content (e.g., a file), but not both.

A root *also* is a document — here, “root” refers to the root of a tree of documents.

Root and Document IDs

Each root has an ID unique to your app to identify that root as being distinct from any other root. This is a string.

Each document — files and directories alike — will have a string document ID to uniquely identify that document within its root. If you have some natural identifier (e.g., a primary key in some table), feel free to use it. Otherwise, you might consider your document ID to be some path to get to the document.

And, since a root is also a document, a root will have both a root ID and a document ID.

These document IDs need to be durable. Clients, or the Storage Access Framework itself, may wind up caching these IDs. Hence, pick something that not only uniquely identifies this document, but will *continue* to uniquely identify the document even after the document has been modified.

Pieces of a Provider

The Documents/Provider sample application implements a document provider, one that serves documents baked into the app itself in `assets/`. Of course, this is a rather artificial scenario — usually, a document provider will be working with a read/write data store, like internal storage.

The Activity

A document provider app probably needs a real UI. Perhaps that UI is for broader functionality that the app provides on top of serving documents. Perhaps that UI is merely to configure the document store, such as providing account credentials for the online storage service that the document provider exposes on Android.

If nothing else, you will need a do-nothing activity for the user to run, to ensure that your app is moved out of the stopped state.

In the sample app, this is handled by `MainActivity`, which uses `Theme.Translucent.NoDisplay` to avoid a UI and just shows a `Toast` to indicate that the provider is now activated:

```
package com.commonsware.android.documents.provider;

import android.app.Activity;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

public class MainActivity extends Activity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Toast.makeText(this, R.string.activated, Toast.LENGTH_LONG).show();
    finish();
  }
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/MainActivity.java)

The API Level Resources

The document provider itself comes courtesy of a subclass of `DocumentsProvider`. However, the `DocumentsProvider` class only exists on API Level 19 and higher — our subclass is useless on older devices. To ensure that our provider is only used on API Level 19 and higher, we should only enable it on API Level 19+ devices.

To that end, in `res/values/bools.xml`, we have a boolean resource named `min19`, set to `false`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="min19">false</bool>
</resources>
```

(from Documents/Provider/app/src/main/res/values/bools.xml)

In `res/values-v19/bools.xml`, we redefine that boolean resource to be `true`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="min19">true</bool>
</resources>
```

(from Documents/Provider/app/src/main/res/values-v19/bools.xml)

Hence, when we refer to the `min19` boolean resource, we will get `true` or `false` depending upon whether we are on API Level 19 or not.

The Manifest

Since `DocumentsProvider` is a subclass of `ContentProvider`, we will need a `<provider>` element in the manifest pointing to our subclass of `DocumentsProvider`:

```
    <provider
      android:name=".DemoDocumentsProvider"
      android:authorities="com.commonsware.android.documents.provider"
      android:enabled="@bool/min19"
      android:exported="true"
      android:grantUriPermissions="true"
      android:permission="android.permission.MANAGE_DOCUMENTS">
      <intent-filter>
        <action android:name="android.content.action.DOCUMENTS_PROVIDER" />
      </intent-filter>
    </provider>
```

(from Documents/Provider/app/src/main/AndroidManifest.xml)

That element:

- Uses our `@bool/min19` resource from above to indicate that this component should only be enabled on API Level 19 and higher
- Is exported, but requires that applications looking to talk to our provider hold the `MANAGE_DOCUMENTS` permission, which can only be held by the firmware (or apps signed with the firmware's signing key)
- Sets the `android:grantUriPermissions` attribute to `true`, as that will be used by `DocumentsProvider` to allow third-party apps limited, conditional access to our documents
- Has your standard `android:name` and `android:authorities` attributes, as with any other `<provider>`

In addition, the `<provider>` has a nested `<intent-filter>` element. This may seem odd, as this used to be impossible, and it is not intuitively obvious what it would mean for a `ContentProvider` to have an `IntentFilter`. It also is not documented as being allowed on `<provider>`, so we have no official explanation of what this means. Most likely, the magic `android.content.action.DOCUMENTS_PROVIDER` filter is being used simply as a marker, to indicate to Android that this particular `<provider>` is part of the Storage Access Framework and implements a `DocumentsProvider`.

The DocumentsProvider

The real business logic of publishing documents comes from your subclass of `DocumentsProvider`. As this class is new to API Level 19, your build target (e.g., `compileSdkVersion` in `build.gradle`) needs to be 19 or higher.

A minimal `DocumentsProvider` implementation will typically need five methods, outlined below.

onCreate()

As with any `ContentProvider`, your `DocumentsProvider` can override `onCreate()` to perform initialization work. Technically, this is not required, but the odds are very good that you will have *something* that you need to initialize.

In the case of our sample `DocumentsProvider` — named `DemoDocumentsProvider` — `onCreate()` simply obtains access to an `AssetManager` instance that can be used for serving documents:

```
private AssetManager assets;

@Override
public boolean onCreate() {
  assets=getContext().getAssets();

  return(true);
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

queryRoots()

Your `queryRoots()` method needs to return information about the root(s) that your provider will provide.

However, rather than returning this in the form of some clean object model (e.g., a `List` of `Document.Root` objects or some such), the return value is a `Cursor`. While in principle this `Cursor` could come from a database, in many cases it will be a `MatrixCursor`, which is a `Cursor` interface over a two-dimensional array representing the rows and columns.

From here, you should return all *presently valid* roots. The "presently valid" part is because a root might exist but not be usable at the present time. For example, suppose that you are writing a `DocumentsProvider` that provides a document interface to an Internet-hosted storage service. In this case, you may need the user to authenticate in order to allow access to those files, such as to pass that authentication data along to the Web service to be able to retrieve directory and file data. If the user is not presently logged in, though, not only can you not talk to the Web service right now, but you do not have the ability to force the user to authenticate right now. Instead, you will have to cull the root(s) governed by those authentication credentials. This may mean that the `Cursor` you return has no rows, as you simply do not have anything that can be published right now.

The `Cursor` that you return will have one row per presently valid root. The columns will be ones defined on the `DocumentsContract.Root` class. Your `queryRoots()` method is passed a `String` array representing the columns requested by the Storage Access Framework. As your app may not support all of those columns, you will need to determine the intersection between the requested columns and the ones you support.

The sample app defines a `SUPPORTED_ROOT_PROJECTION` static data member to list the `DocumentsContract.Root` columns that are supported in general:

```
private static final String[] SUPPORTED_ROOT_PROJECTION=new String[] {
    Root.COLUMN_ROOT_ID, Root.COLUMN_FLAGS, Root.COLUMN_TITLE,
    Root.COLUMN_DOCUMENT_ID, Root.COLUMN_ICON };
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

And the demo provider has a private `netProjection()` utility method that computes the intersection between the requested columns and the supported ones:

```
private static String[] netProjection(String[] requested, String[] supported) {
  if (requested==null) {
    return(supported);
  }

  ArrayList<String> result=new ArrayList<String>();

  for (String request : requested) {
    for (String support : supported) {
      if (request.equals(support)) {
        result.add(request);
        break;
      }
    }
  }

  return(result.toArray(new String[0]));
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

That net projection is used in the `MatrixCursor` constructor, to teach it the available columns, as part of the `queryRoots()` implementation:

```
@Override
public Cursor queryRoots(String[] projection)
    throws FileNotFoundException {
  String[] netProjection=
      netProjection(projection, SUPPORTED_ROOT_PROJECTION);
  MatrixCursor result=new MatrixCursor(netProjection);
  MatrixCursor.RowBuilder row=result.newRow();

  row.add(Root.COLUMN_ROOT_ID, ROOT_ID);
  row.add(Root.COLUMN_ICON, R.drawable.ic_launcher);
  row.add(Root.COLUMN_FLAGS, Root.FLAG_LOCAL_ONLY);
  row.add(Root.COLUMN_TITLE, getContext().getString(R.string.root));
  row.add(Root.COLUMN_DOCUMENT_ID, ROOT_DOCUMENT_ID);
```

```
    return(result);
  }
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

`queryRoots()` then adds a row to the `MatrixCursor`, through a `MatrixCursor.RowBuilder`, containing five columns:

1. `DocumentsContract.Root.COLUMN_ROOT_ID` is the root ID for this root, as described earlier in this chapter.
2. `DocumentsContract.Root.COLUMN_ICON`, which is a reference to a drawable resource that may be used in Storage Access Framework UI to help visually represent this root. In principle, this could be anything; in practice, you will probably choose your launcher icon, as it is the icon that the user will recognize.
3. `DocumentsContract.Root.COLUMN_FLAGS`, indicating which optional capabilities this root supports. In this case, the only flag we are setting is `FLAG_LOCAL_ONLY`, indicating that network I/O is not required to browse the contents of the provider. Our sample app indicates that it is local-only, as its documents are all packaged in `assets/`. A provider backed by a Web service, though, would not include this flag, so the Storage Access Framework knows that calls to some of the other methods (e.g., `queryChildDocuments()`) may take a significant amount of time.
4. `DocumentsContract.Root.COLUMN_TITLE`, which is a string identifying this root. The title and icon will tend to be included in Storage Access Framework-supplied UIs. In this case, with only a single root, the title is hard-coded to be a string resource. In other cases, this might be some other human-grokkable display name (e.g., the name of some storage service account).
5. `DocumentsContract.Root.COLUMN_DOCUMENT_ID`, which returns the document ID representing the document tree for this root.

In this case, the document IDs for this `DocumentsProvider` are the relative paths within `assets/` of the files, starting from a root `docs/` directory. So, while the root ID could be anything, the root document ID should be consistent with the other document ID values. In this case, the sample app uses:

```
private static final String ROOT_ID="thisIsMyBoomstick";
private static final String ROOT_DOCUMENT_ID="docs";
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

queryChildDocuments()

As noted previously, some documents will represent a directory, while others will represent files. For those that represent a directory, `queryChildDocuments()` will need to return the document information for the contents of the directory.

`queryChildDocuments()` is passed:

- the document ID of the directory
- the columns, defined on the `DocumentsContract.Document` class, that the Storage Access Framework wants
- the sort order, expressed as a SQL-style `ORDER BY` clause (minus the actual `ORDER BY` part), that you might use to help control the order in which to return the child documents (or ignore if you wish)

As with `queryRoots()`, we need to come up with the intersection of the columns that the requester asks for and the columns that we support. There is a `static` string array named `SUPPORTED_DOCUMENT_PROJECTION` that represents the columns that we support:

```
private static final String[] SUPPORTED_DOCUMENT_PROJECTION=
    new String[] { Document.COLUMN_DOCUMENT_ID, Document.COLUMN_SIZE,
        Document.COLUMN_MIME_TYPE, Document.COLUMN_DISPLAY_NAME,
        Document.COLUMN_FLAGS};
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

The `queryChildDocuments()` method then uses the same `netProjection()` helper method that `queryRoots()` did to determine the intersection:

```
@Override
public Cursor queryChildDocuments(String parentDocId,
                                  String[] projection,
                                  String sortOrder)
    throws FileNotFoundException {
  String[] netProjection=
      netProjection(projection, SUPPORTED_DOCUMENT_PROJECTION);
  MatrixCursor result=new MatrixCursor(netProjection);

  try {
    String[] children=assets.list(parentDocId);

    for (String child : children) {
      addDocumentRow(result, child,
```

```
                      parentDocId + File.separator + child);
    }
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(),
          "Exception reading asset dir", e);
  }

  return(result);
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

As with `queryRoots()`, the return value of `queryChildDocuments()` is a `Cursor` representing the documents contained in this directory. Once again, we use a `MatrixCursor` to build up an in-memory `Cursor`, this time for all files within the `assets/` directory denoted by `parentDocId`, using the `list()` method on `AssetManager` to find out what those files are.

The logic to populate the `MatrixCursor` is delegated to an `addDocumentRow()` private method, as we will be using it elsewhere in this `DocumentsProvider` implementation. `addDocumentRow()` creates a `MatrixCursor.RowBuilder` and fills in the supported columns:

```
private void addDocumentRow(MatrixCursor result, String child,
                            String assetPath) throws IOException {
  MatrixCursor.RowBuilder row=result.newRow();

  row.add(Document.COLUMN_DOCUMENT_ID, assetPath);

  if (isDirectory(assetPath)) {
    row.add(Document.COLUMN_MIME_TYPE, Document.MIME_TYPE_DIR);
  }
  else {
    String ext=MimeTypeMap.getFileExtensionFromUrl(assetPath);

    row.add(Document.COLUMN_MIME_TYPE,
        MimeTypeMap.getSingleton().getMimeTypeFromExtension(ext));
    row.add(Document.COLUMN_SIZE, getAssetLength(assetPath));
  }

  row.add(Document.COLUMN_DISPLAY_NAME, child);
  row.add(Document.COLUMN_FLAGS, 0);
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

Of note:

- the document ID of the child is simply its relative path within `assets/`
- the MIME type is a special one if the child document represents a directory, or else is looked up using `MimeTypeMap` if the child document represents a file
- the "display name" could be something special (e.g., the `<title>` of a Web page), but in this case is just the filename

To determine if an asset path represents a directory, the `isDirectory()` utility method just sees if `list()` returns a non-empty list:

```
private boolean isDirectory(String assetPath) throws IOException {
  return(assets.list(assetPath).length>=1);
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

To find the size of a document — to fill in the `COLUMN_SIZE` column in the output — we can ask the `AssetManager` for a `FileDescriptor` on the asset, then obtain the length from that descriptor, as seen in the `getAssetLength()` utility method:

```
private long getAssetLength(String assetPath) throws IOException {
  return(assets.openFd(assetPath).getLength());
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

The net result is that, given the name of a directory in `assets/`, we return a `Cursor` with one row per child of that directory, with columns indicating details of that child.

queryDocument()

`queryDocument()` is similar to `queryChildDocuments()`. Both return a `Cursor` with the same sorts of columns as output. The difference: `queryDocument()` provides you with the document ID of a file, and you return details of that file. By contrast, `queryChildDocuments()` gives you the document ID of a directory, and you return the details of all documents within that directory.

This is why `addDocumentRow()` was implemented as a separate method, as we need the same business logic (populate a `MatrixCursor` row based on an asset path) from `queryDocument()`:

```
@Override
public Cursor queryDocument(String documentId, String[] projection)
    throws FileNotFoundException {
  String[] netProjection=
      netProjection(projection, SUPPORTED_DOCUMENT_PROJECTION);
  MatrixCursor result=new MatrixCursor(netProjection);

  try {
    addDocumentRow(result, Uri.parse(documentId).getLastPathSegment(),
        documentId);
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(), "Exception reading asset dir", e);
  }

  return(result);
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

In this case, the only thing different is that we need to get the bare filename, for use in the `DISPLAY_NAME` field. Here, we cheat a bit and use `getLastPathSegment()` on `Uri` to obtain the filename.

openDocument()

The `openDocument()` method behaves much like the `openFile()` method of a classic streaming `ContentProvider`: given a path, you return a `ParcelFileDescriptor` representing the file contents. For documents that are true files on the filesystem, you can use the static `open()` method on `ParcelFileDescriptor`. For documents that are not files on the filesystem — such as documents that are assets in the APK — you will need to set up a `ParcelFileDescriptor` pipe and stream the content that way.

That is what `DemoDocumentsProvider` does, using logic copied from the book's streaming `ContentProvider` samples:

```
@Override
public ParcelFileDescriptor openDocument(String documentId,
                                         String mode,
                                         CancellationSignal signal)
    throws FileNotFoundException {
  ParcelFileDescriptor[] pipe=null;

  try {
```

```
      pipe=ParcelFileDescriptor.createPipe();
      AssetManager assets=getContext().getAssets();

      new TransferThread(assets.open(documentId),
          new ParcelFileDescriptor.AutoCloseOutputStream(pipe[1])).start();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception opening pipe", e);
      throw new FileNotFoundException("Could not open pipe for: "
          + documentId);
    }

    return(pipe[0]);
  }
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

`openDocument()` is passed three parameters:

1. The document ID of the document to stream back
2. A file mode (`r`, `w`, or `wt`) indicating what sort of operations the client wants to perform on the stream
3. A `CancellationSignal` that we can use to find out that our streaming is being interrupted

In this case:

- `openDocument()` ignores the `mode`, because it did not return `FLAG_SUPPORTS_WRITE` in either `queryChildDocuments()` or `queryDocument()` to indicate that writing is an option, so the `mode` should always be `r`
- `openDocument()` ignores the `CancellationSignal`, though in reality it should pay attention to it when streaming back the content and stop streaming when requested

The `TransferThread` that does the actual streaming is, once again, the same as the one used earlier in this book for a streaming `ContentProvider`:

```
  static class TransferThread extends Thread {
    InputStream in;
    OutputStream out;

    TransferThread(InputStream in, OutputStream out) {
      this.in=in;
      this.out=out;
    }
```

```
  @Override
  public void run() {
    byte[] buf=new byte[8192];
    int len;

    try {
      while ((len=in.read(buf)) >= 0) {
        out.write(buf, 0, len);
      }

      in.close();
      out.flush();
      out.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
          "Exception transferring file", e);
    }
  }
}
```

(from Documents/Provider/app/src/main/java/com/commonsware/android/documents/provider/DemoDocumentsProvider.java)

The Results

If you have both this sample app and the one from the previous chapter, then run the one from the previous chapter to bring up the Storage Access Framework UI, you will see our provider among the list of available providers:

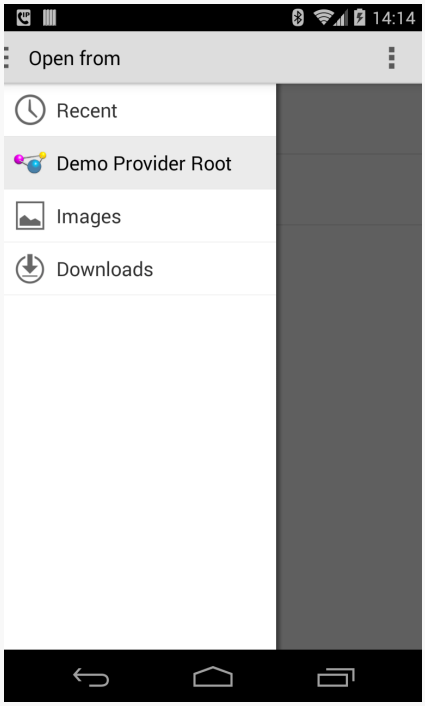


Figure 794: Storage Access Framework Picker, Showing Custom Provider

The provider's `assets/docs/` directory contains three files, one just off the root and two in a `bar/` subdirectory:

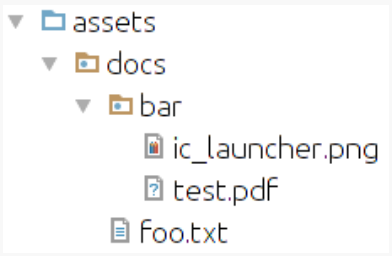


Figure 795: DocumentsProvider Sample Documents

Hence, tapping on our provider in the Storage Access Framework picker brings up the contents of the root document:

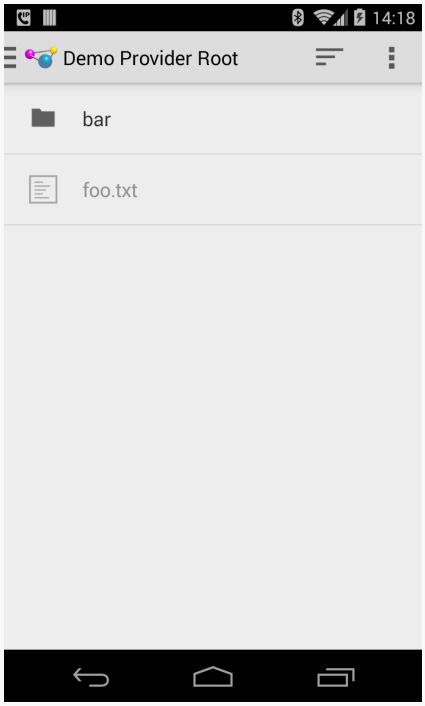


Figure 796: Storage Access Framework Picker, Showing Documents in Root

Tapping on the `bar/` directory brings up its contents in turn:

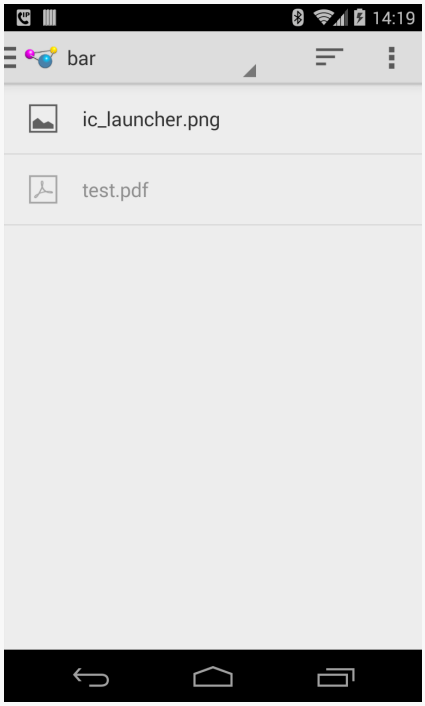


Figure 797: Storage Access Framework Picker, Showing Yet More Documents

Tapping on one of the files brings up the details for that file:

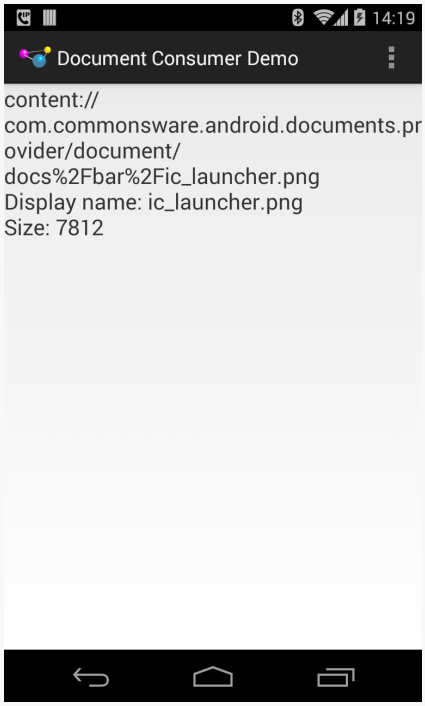


Figure 798: Document Consumer, Showing Details of Picked Document

Optional Provider Capabilities

A `DocumentsProvider` can do a fair bit more than what the above sample app demonstrates. While the sample will suffice for the basics, it is reasonably likely that a production-grade `DocumentsProvider` will need to implement and provide some other optional capabilities, such as those described in this section.

Other CRUD Operations

CRUD — Create, Read, Update, and Delete — is a standard shorthand for the basic operations one can perform on data. The sample app handles the "Read" portion of CRUD, but a `DocumentsProvider` can support all of them if desired.

Create

It may be that your `DocumentsProvider` is only going to serve up documents that were created in your app, or were created outside of the Android device (e.g., on a Web app). If, however, you want consumers of your provider to be able to use

ACTION_CREATE_DOCUMENT to create new documents in your provider, you will need to do a few things.

First, in the COLUMN_FLAGS for the relevant root(s) returned by queryRoots(), you will need to include FLAG_SUPPORTS_CREATE, defined on DocumentsContract.Root. This indicates that at least one directory within that root supports creating new documents. Without this flag, your root(s) will be shown for ACTION_OPEN_DOCUMENT requests but not ACTION_CREATE_DOCUMENT requests.

Next, in one or more directories returned as part of queryDocument() and queryChildDocuments() calls, in the COLUMN_FLAGS column, you will need to include FLAG_DIR_SUPPORTS_CREATE, defined on DocumentsContract.Documents. This indicates that this document is a directory that supports creating new documents inside of it. Otherwise, a directory will be assumed to not support creating new documents. Note that this flag is only used for documents representing directories, not documents representing files.

Finally, you will need to implement createDocument() in your DocumentsProvider. This will be called if a consumer app used ACTION_CREATE_DOCUMENT and the user chose your provider and one of your directories for the new document. You are passed in:

- the document ID of the directory
- the MIME type of the new file
- a suggested display name to use for the new file, though you can modify this if needed

Your job, in createDocument(), is to create the document and return the document ID for the newly-created document. For example, if your documents are held in internal storage, you might create a new file for the document itself plus a database row in some documents table to hold the MIME type and display name.

Update

For something like files, an "update" is replacing the current contents with something new. In the case of a streaming protocol like DocumentsProvider, this implies that your provider can support output as well as input.

This too requires a few changes to your provider.

First, for the file(s) that can be updated, in the results for `queryDocument()` and `queryChildDocuments()`, you will need to include `FLAG_SUPPORTS_WRITE` in `COLUMN_FLAGS`, to indicate that writing to this file should work.

Then, you will need to pay attention to the `mode` passed into `openDocument()`. If the mode is `w` or `wt`, you would need to arrange to support writing the file, where your background thread reads from an `InputStream` on the pipe and writes the data to wherever your data is being stored.

Delete

For any documents that the consumer can delete, include `FLAG_SUPPORTS_DELETE` in `COLUMN_FLAGS` in the results for `queryDocument()` and `queryChildDocuments()`.

You will also need to implement `deleteDocument()` in your `DocumentsProvider`. You are supplied the document ID to delete, and your job is to delete it.

If the document represents a directory, you may also need to delete all of its children. That really depends on how you are leveraging the "directory" construct in `DocumentsProvider`:

- If the "directory" is like a filesystem directory, where children have only one parent, you will want to delete the children when you delete the parent
- If the "directory" is more like a category, such as a tag, where children could have multiple parents, you will need to decide how to handle the children that would be orphaned by your deleting the last parent (delete the children? move them to some other default parent? something else?)

Change Notification

If the data served by your provider changes, it is incumbent upon you to let possible consumers know about the change. For example, if you elect to delete children when you delete their parent, you should let consumers know that those children were deleted. This is not necessary for direct operations performed by consumers (e.g., writing to a document), but is necessary for anything else.

To do that, you call `notifyChange()` on a `ContentResolver`, just as you would for changes to the data in a `ContentProvider`. However, `notifyChange()` takes a `Uri` as a parameter, to indicate the scope of the change. There are `static` utility methods on `DocumentsContract` that will return a `Uri` that you can use. Notably:

- `buildDocumentUri()`, given your authority and a document ID, provides a `Uri` that points to that document
- `buildChildDocumentsUri()`, given your authority and a document ID, provides a `Uri` that represents the collection of children of that document

So, for example, if by deleting a parent you also delete the children, you would use `buildChildDocumentsUri()` with `notifyChange()` to ensure that consumers know that those children were modified. The Storage Access Framework will use methods like `queryChildDocuments()` to determine that the children were deleted in this case.

Thumbnails

By default, the Storage Access Framework will use stock icons for directories and files. You can supply your own thumbnails instead, though, if you want. To do this:

- Include `FLAG_SUPPORTS_THUMBNAIL` in the `COLUMN_FLAGS` for the affected document(s) in `queryDocument()` and `queryChildDocuments()`
- Implement `openDocumentThumbnail()` in your `DocumentsProvider`

`openDocumentThumbnail()` is provided the document ID of the document whose thumbnail is required, along with a `Point` object providing a requested size. While your thumbnail does not have to exactly match that size — for example, the aspect ratio that is requested may not match the thumbnail — it should be close.

However, the return value for `openDocumentThumbnail()` is an `AssetFileDescriptor`, which is a wrapper around a `ParcelFileDescriptor`. If your image exists as a file that happens to be the right size, return it by using the static `open()` method on `ParcelFileDescriptor` is fairly straightforward. If, however, you need to scale your source image to fit the desired size, implementing this via a pipe will be moderately tedious.

Recent Documents

If your app has its own concept of recent documents, you can expose that roster to the Storage Access Framework, which can incorporate it as part of its UI. To do this:

- Have your `queryRoots()` method include `FLAG_SUPPORTS_RECENTS` in the `COLUMN_FLAGS` value for the root(s) that support recent documents
- Implement `queryRecentDocuments()` on your `DocumentsProvider`, where you are given the root ID (*not* a document ID!) of one of your roots, and you

need to return the same sort of `Cursor` as you would from `queryChildDocuments()`, but representing the recent documents for that root

The two constraints upon the returned `Cursor` are:

- It should be sorted descending based on last-modified date (e.g., the `COLUMN_LAST_MODIFIED` column in your rows)
- It should be capped at 64 rows, though it can be less if desired

Note that you can, if you wish, have the `Cursor` return rows reflecting both files and directories — you are not limited to one or the other.

Search

If your provider has its own search capability, you can expose that to the Storage Access Framework, which in turn can make it available to users looking for a certain document. To support this:

- Have your `queryRoots()` method include `FLAG_SUPPORTS_SEARCH` in the `COLUMN_FLAGS` value for the root(s) that support searching
- Implement `querySearchDocuments()` on your `DocumentsProvider`, where you are given the root ID (*not* a document ID!) of one of your roots, and you need to return the same sort of `Cursor` as you would from `queryChildDocuments()`, but representing the results of a search

`querySearchDocuments()` is passed a `String` representing the search expression entered by the user. It is up to you to decide what that expression means. It is also up to you to determine where you are searching for that expression (filenames? file contents?).

Note that the `Cursor` you return should only contain documents that reflect files, not documents that point to directories.

Other Flags

There are a few other flags that are available to you on `DocumentsContract.Document` that you can use in `COLUMN_FLAGS` for `Cursor` results representing a document or collection of documents, such as the results of `queryDocument()` and `queryChildDocuments()`:

- If the document represents a directory, and you are supporting thumbnails, and you would like the contents of this directory to be represented in a thumbnail grid as opposed to a list, include `FLAG_DIR_PREFERS_GRID`
- If the document represents a directory, and you feel that users will be better served showing the documents in descending order based upon `COLUMN_LAST_MODIFIED`, rather than alphabetical by display name, include `FLAG_DIR_PREFERS_LAST_MODIFIED`

Encrypted Storage

SQLite databases, by default, are stored on internal storage, accessible only to the app that creates them.

At least, that is the theory.

In practice, it is conceivable that others could get at an app's SQLite database, and that those "others" may not have the user's best interests at heart. Hence, if you are storing data in SQLite that should remain confidential despite extreme measures to steal the data, you may wish to consider encrypting the database.

Perhaps the simplest way to encrypt a SQLite database is to use SQLCipher. SQLCipher is a SQLite extension that encrypts and decrypts database pages as they are written and read. However, SQLite extensions need to be compiled into SQLite, and the stock Android SQLite does not have the SQLCipher extension.

SQLCipher for Android, therefore, comes in the form of a replacement implementation of SQLite that you add as an NDK library to your project. It also ships with replacement editions of the `android.database.sqlite.*` classes that use the SQLCipher library instead of the built-in SQLite. This way, your app can be largely oblivious to the actual database implementation, particularly if it is hidden behind a `ContentProvider` or similar abstraction layer.

SQLCipher for Android is a joint initiative of Zetetic (the creators of SQLCipher) and the Guardian Project (home of many privacy-enhancing projects for Android). SQLCipher for Android is open source, under the Apache License 2.0.

Prerequisites

Understanding this chapter requires that you have read the chapter on database access.

Scenarios for Encryption

So, why might you want to encrypt a database?

Some developers probably are thinking that this is a way of protecting the app's content against "those pesky rooted device users". In practice, this is unlikely to help. As with most encryption mechanisms, SQLCipher uses an encryption key. If the app has the key, such as being hard-coded into the app itself, anyone can get the key by reverse-engineering the app.

Rather, encrypted databases are to help the user defend their data against other people seeing it when they should not. The classic example is somebody leaving their phone in the back of a taxi — if that device winds up in the hands of some group with the skills to root the device, they can get at any unencrypted content they want. While some users will handle this via the whole-disk encryption available since Android 3.0, others might not.

If the database is going anywhere other than internal storage, there is all the more reason to consider encrypting it, as then it may not even require a rooted device to access the database. Scenarios here include:

1. Databases stored on external storage
2. Databases backed up using external storage, BackupManager, or another Internet-based solution
3. Databases explicitly being shared among a user's devices, or between a user's device and a desktop (note that SQLCipher works on many operating systems, including desktops and iOS)

Obtaining SQLCipher

SQLCipher is available from Zetitec. As of July 2016, the current shipping version was 3.5.0. **It is very important for you to use 3.5.0 or higher**, as earlier versions of SQLCipher for Android will not work on Android 7.0 or higher versions of Android.

In Android Studio, to add SQLCipher for Android to your project, just add the official AAR dependency:

```
dependencies {
    implementation 'net.zetetic:android-database-sqlcipher:3.5.0@aar'
}
```

Using SQLCipher

If you have existing code that uses classic Android SQLite, you will need to change your import statements to pick up the SQLCipher for Android equivalents of the classes. For example, you obtain `SQLiteDatabase` now from `net.sqlcipher.database.sqlcipher`, not `android.database.sqlite`. Similarly, you obtain `SQLException` from `net.sqlcipher.database` instead of `android.database`. Unfortunately, there is no complete list of which classes need this conversion — `Cursor`, for example, does not. Try converting everything from `android.database` and `android.database.sqlite`, and leave alone those that do not exist in the SQLCipher for Android equivalent packages.

Before starting to use SQLCipher for Android, you need to call `SQLiteDatabase.loadLibs()`, supplying a suitable `Context` object as a parameter. This initializes the necessary libraries. If you are using a `ContentProvider`, just call this in `onCreate()` before actually using anything else with your database. If you are not using a `ContentProvider`, you probably will want to create a custom subclass of `Application` and make this call from that class' `onCreate()`, and reference your custom `Application` class in the `android:name` attribute of the `<application>` element in your manifest. Either of these approaches will help ensure that the libraries are ready before you try doing anything with the database.

Finally, when calling `getReadableDatabase()` or `getWritableDatabase()` on `SQLiteDatabase`, you need to supply the encryption key to use. For the purposes of book examples, a hard-coded passphrase is sufficient. However, those can be trivially reverse-engineered, and so they offer little real-world protection. But, they keep the code simple, which is useful when examining APIs.

The `Database/ConstantsSecure-AndroidStudio` sample app is yet another variation of the `ConstantsBrowser` sample that we have been using for most of the database examples. From the standpoint of the `ConstantsBrowser` activity and `ConstantsFragment` UI, nothing is different. However, `DatabaseHelper` uses SQLCipher, rather than SQLite.

In the `DatabaseHelper` constructor, we call `loadLibs()` on the `SQLiteDatabase` class, which is a required initialization step to get the native libraries set up:

```
public DatabaseHelper(Context context) {
  super(context, DATABASE_NAME, null, SCHEMA);

  SQLiteDatabase.loadLibs(context);
}
```

(from Database/ConstantsSecure-AndroidStudio/app/src/main/java/com/commonsware/android/sqlcipher/DatabaseHelper.java)

It also offers zero-argument `getReadableDatabase()` and `getWritableDatabase()` methods, akin to those offered by the regular `SQLiteOpenHelper`. However, the `DatabaseHelper` editions turn around and invoke the one-argument equivalents on the SQLCipher edition of `SQLiteOpenHelper`:

```
SQLiteDatabase getReadableDatabase() {
  return(super.getReadableDatabase(PASSPHRASE));
}

SQLiteDatabase getWritableDatabase() {
  return(super.getWritableDatabase(PASSPHRASE));
}
```

(from Database/ConstantsSecure-AndroidStudio/app/src/main/java/com/commonsware/android/sqlcipher/DatabaseHelper.java)

Here, the `PASSPHRASE` is just a hard-coded string:

```
private static final String PASSPHRASE=
    "hard-coding passphrases is only for sample code;"+
    "nobody does this in production";
```

(from Database/ConstantsSecure-AndroidStudio/app/src/main/java/com/commonsware/android/sqlcipher/DatabaseHelper.java)

That is all the changes that are needed to use SQLCipher.

SQLCipher Limitations

Alas, SQLCipher for Android is not perfect.

It will add a few MB to the size of your APK file per CPU architecture. For most modern Android devices, this extra size will not be a huge issue, though it will be an impediment for older devices with less internal storage, or for apps that are getting close to the size limits imposed by the Play Store or other distribution mechanisms.

The chapter on the NDK contains a section about a technology called `libhoudini` that can help reduce this bloat, albeit with a significant performance penalty.

However, the size is mostly from code, and that may cause a problem for Eclipse users. Eclipse may crash with its own `OutOfMemoryError` during the final build process. To address that, find your `eclipse.ini` file (location varies by OS and installation method) and increase the `-Xmx` value shown on one of the lines (e.g., change it to `-Xmx512m`).

Other code that expects to be using native SQLite databases will require alteration to work with SQLCipher for Android databases. For example, the `SQLiteAssetHelper` described elsewhere in this book would need to be ported to use the SQLCipher for Android implementations of `SQLiteOpenHelper`, `SQLiteDatabase`, etc. This is not too difficult for an open source component like `SQLiteAssetHelper`.

Passwords and Sessions

Given an encrypted database, there are several ways that an attacker can try to access the data, including:

1. Use a brute-force attack via the app itself
2. Use a brute-force attack on the database directly, by copying it to some other machine
3. Obtain the password by the strategic deployment of a $5 wrench

The classic way to prevent the first approach is by having business logic that prevents lots of failed login attempts in a short period of time. This can be built into your login dialog (or the equivalent), tracking the number and times of failed logins and introducing delays, forced app exits, or something to add time and hassle for trying lots of passwords.

Since manually trying passwords is nasty, brutish, and long, many attackers would automate the process by copying the SQLCipher database to another machine (e.g., desktop) and running a brute-force attack on it directly. SQLCipher for Android has many built-in protections to help defend against this. So long as you are using a sufficiently long and complex encryption key, you should be fairly well-protected against such attacks.

Defending against wrenches is decidedly more difficult and is beyond the scope of this book.

About Those Passphrases…

Having a solid encryption algorithm, like the AES-256 used by default with SQLCipher for Android, is only half the battle. The other half is in using a high-quality passphrase, one that is unlikely to be guessed by anyone looking to break the encryption.

Upgrading to Encryption

Suppose you have an app already out on the market, and you decide that you want to add the option for encryption. It is fairly likely that the user will be miffed if they lose all their data in the process of switching to an encrypted database. Therefore, you will want to try to retain their data.

SQLCipher for Android does not support in-place encryption of database. However, it *does* support working with unencrypted databases and encrypted databases simultaneously, giving you the option of migration.

The approach boils down to:

- Open the unencrypted database in SQLCipher for Android, using an empty passphrase
- Use the `ATTACH` statement to open the encrypted database inside the same SQLCipher for Android session
- Use a supplied `sqlcipher_export()` function to migrate most of the data
- Copy the Android database schema version between the databases
- `DETACH` the encrypted database
- Close the unencrypted database (and, presumably, delete it)
- Use the encrypted database from this point forward

Since both database files will exist at one time, you will find it simplest to use separate names for them (e.g., `stuff.db` and `stuff-encrypted.db`).

To see how this works, take a look at the [`Database/SQLCipherPassphrase-AndroidStudio`], which is a variation of the original, non-`ContentProvider` "constants" sample app, this time using SQLCipher for Android and supporting an upgrade from a non-encrypted database to an encrypted one.

The bulk of the logic for handling the encryption upgrade is in a static `encrypt()` method on our `DatabaseHelper`:

```
static void encrypt(Context ctxt) {
  SQLiteDatabase.loadLibs(ctxt);

  File dbFile=ctxt.getDatabasePath(DATABASE_NAME);
  File legacyFile=ctxt.getDatabasePath(LEGACY_DATABASE_NAME);

  if (!dbFile.exists() && legacyFile.exists()) {
    SQLiteDatabase db=
        SQLiteDatabase.openOrCreateDatabase(legacyFile, "", null);

    db.rawExecSQL(String.format("ATTACH DATABASE '%s' AS encrypted KEY '%s';",
                                dbFile.getAbsolutePath(), PASSPHRASE));
    db.rawExecSQL("SELECT sqlcipher_export('encrypted')");
    db.rawExecSQL("DETACH DATABASE encrypted;");

    int version=db.getVersion();

    db.close();

    db=SQLiteDatabase.openOrCreateDatabase(dbFile, PASSPHRASE, null);
    db.setVersion(version);
    db.close();

    legacyFile.delete();
  }
}
```

(from Database/SQLCipherPassphrase-AndroidStudio/app/src/main/java/com/commonsware/android/constants/DatabaseHelper.java)

First, we initialize SQLCipher for Android by calling `loadLibs()` on the SQLCipher version of `SQLiteDatabase`. We could do this someplace else, but for this sample, this is as good a spot as any.

We then create `File` objects pointing at the locations of the old, unencrypted database (with a name represented by a `LEGACY_DATABASE_NAME` static data member) and the new encrypted database (`DATABASE_NAME`). To get the `File` locations of those databases, we use `getDatabasePath()`, a method on `Context`, which returns the correct location for a database file given its name.

If the encrypted database exists, there is nothing that we need to do. Similarly, if it does not exist but the unencrypted database *also* does not exist, there is nothing that we *can* do. In either of those cases, we skip over the rest of the logic. In the first case, we already did the conversion (presumably); in the latter case, this is a new installation, and our `SQLiteOpenHelper` `onCreate()` logic will handle that. But, in the case where we do not have the encrypted database but do have the unencrypted one, we can create the encrypted database from the unencrypted data, which is what the bulk of the `encrypt()` method does.

To that, we:

- Use `openOrCreateDatabase()` to open the already-existing unencrypted database file *in SQLCipher for Android*, using `""` as the passphrase.
- Use a `rawExecSQL()` method available on the SQLCipher for Android version of `SQLiteDatabase` to `ATTACH` the encrypted database, given its path, to our database session, using the supplied passphrase. This means that we can access the tables from both databases simultaneously, though we need to prefix all references to the attached database via its handle, `encrypted`.
- Use `rawExecSQL()` to execute `SELECT sqlcipher_export('encrypted')`, which copies most of our data from the unencrypted database (the database we have open) into the `encrypted` database (the one we attached). The big thing that `sqlcipher_export()` does *not* copy is the schema version number that Android maintains.
- Use `rawExecSQL()` to `DETACH` the attached `encrypted` database, as we no longer need it.
- Call `getVersion()` on the `SQLiteDatabase` representing the unencrypted database, to retrieve the schema version number that Android maintains.
- Close the unencrypted database and open the encrypted one using `openOrCreateDatabase()`.
- Use `setVersion()` on `SQLiteDatabase` to set the schema version of the encrypted database to the value we had from the unencrypted database.
- Close the encrypted database and delete the unencrypted database file. Note that on API Level 16+, we could use the `deleteDatabase()` method on `SQLiteDatabase` to cleanly delete everything associated with SQLite.

The combination of doing all of that migrates our data from an unencrypted database to an encrypted one.

Then, we simply need to call `encrypt()` before we try loading our constants, from `doInBackground()` of our `LoadCursorTask`:

```
private class LoadCursorTask extends BaseTask<Void> {
  private final Context ctxt;

  LoadCursorTask() {
    this.ctxt=getActivity().getApplicationContext();
  }

  @Override
  protected Cursor doInBackground(Void... params) {
    DatabaseHelper.encrypt(ctxt);
```

```
      return(doQuery());
    }
  }
```

(from Database/SQLCipherPassphrase-AndroidStudio/app/src/main/java/com/commonsware/android/constants/ConstantsFragment.java)

To test this upgrade logic, you will need to:

- Run the original unencrypted version of this sample, found in the `Database/Constants` sample application
- Add a new constant using the unencrypted version of the app
- Run the encrypted version of the sample from this section, which shares the same package name as the original and therefore will replace it on your emulator

You will see your added constant appear along with all of the standard ones, yet if you examine `/data/data/com.commonsware.android.constants/databases` on your ARM emulator via DDMS, you will see that your database is now named `constants-crypt.db` instead of `constants.db`, as we have replaced the unencrypted database with an encrypted one.

Changing Encryption Passphrases

Another thing the user might wish to do is change their passphrase. Perhaps they fear that their existing passphrase has been compromised (e.g., a narrow escape from a $5 wrench). Perhaps they rotate their passphrases as a matter of course. Perhaps they simply keep typing in their current one incorrectly and want to switch to one they think they can enter more accurately.

SQLCipher for Android supports a `rekey PRAGMA` that can accomplish this. Given an open encrypted database `db` — opened using the old passphrase – you can change the password to a `newPassword` string variable via:

```
db.execSQL(String.format("PRAGMA rekey = '%s'", newPassword));
```

Note that this may take some time, as SQLCipher for Android needs to re-encrypt the entire database.

Dealing with the Version 3.0.x Upgrade

If you are starting with SQLCipher for Android with the 3.0.x release, all is good.

If you have been using SQLCipher for Android from previous releases, but you are still in development mode, all is still good, so long as you can wipe out your old databases.

If you have apps in production using SQLCipher for Android from previous releases, you will have a small headache: the database structure has changed. SQLCipher for Android provides us with a `PRAGMA cipher_migrate` that we can run to upgrade the database in place to the new structure, once we have opened the database with our passphrase. However:

1. There is no great built-in place to put the code for calling this pragma
2. You do not want to blindly call this pragma every time you open the database, as it results in extra processing time

SQLCipher for Android, in an attempt to help with this, offers a modified version of methods like `openOrCreateDatabase()` on `SQLiteDatabase`, ones that take a `SQLiteDatabaseHook` implementation as the last parameter. This interface requires two methods:

1. `preKey()`, called after the database is opened but before the passphrase is applied
2. `postKey()`, called after the database is opened and after the passphrase is applied, but before anything else is done (e.g., standard `SQLiteOpenHelper` schema version checking)

Both methods are passed the `SQLiteDatabase` as a parameter, for you to do with as needed. So, for example, you could have a `postKey()` implementation that does the `postKey()` call only if needed:

```
public class SQLCipherV3Hook implements SQLiteDatabaseHook {
  private static final String PREFS=
      "net.sqlcipher.database.SQLCipherV3Helper";

  public static void resetMigrationFlag(Context ctxt, String dbPath) {
    SharedPreferences prefs=
        ctxt.getSharedPreferences(PREFS, Context.MODE_PRIVATE);
    prefs.edit().putBoolean(dbPath, false).commit();
  }

  @Override
  public void preKey(SQLiteDatabase database) {
    // no-op
  }
```

```
  @Override
  public void postKey(SQLiteDatabase database) {
    SharedPreferences prefs=
        getContext().getSharedPreferences(PREFS, Context.MODE_PRIVATE);
    boolean isMigrated=prefs.getBoolean(database.getPath(), false);

    if (!isMigrated) {
      database.rawExecSQL("PRAGMA cipher_migrate;");
      prefs.edit().putBoolean(database.getPath(), true).commit();
    }
  }
}
```

You can also pass a `SQLiteDatabaseHook` implementation into the `SQLiteOpenHelper` constructor as the fifth parameter, which will be used when `SQLiteOpenHelper` works with the underlying `SQLiteDatabase`.

Multi-Factor Authentication

Another way to effectively boost the strength of your security is to implement your own multi-factor authentication. In this case, the passphrase is not obtained solely through the user typing in the whole thing, but instead is synthesized from two or more sources. So, in addition to some `EditText` widget for entering in a portion of the passphrase, the rest could come from things like:

- A value written to an NFC tag that the user must tap
- A value encoded in a QR code that the user must scan
- A value obtained by some Bluetooth-connected device via a custom protocol

You, in code, would concatenate the pieces together, possibly using delimiters that cannot be typed in (e.g., ASCII characters below 32) to denote the sources of each segment of the passphrase. The result would be the actual passphrase you would use with SQLCipher for Android.

The objective is to make it easier for users to have more complex passphrases, while not having to type in something complex every time. Tapping an NFC tag is much faster than tapping out a passphrase on a typical phone keyboard, for example. Also, the “something you know and something you have” benefit of multi-factor authentication can help with defending against $5 wrench attacks: if the NFC tag was destroyed, and the user never knew the portion of the passphrase stored on it, the user cannot divulge it.

Of course, this adds risks, such as the NFC tag being destroyed accidentally (e.g., "my dog ate it"). This can be mitigated in some cases by some "admin" being able to reset the password or supply a new NFC tag. In that case, getting the credentials requires *two* kidnappings and *two* $5 wrenches (or the serial application of a single $5 wrench, if budgets preclude buying two such wrenches), adding to the degree of difficulty for breaking the encryption by that means.

Detecting Failed Logins

If you try to decrypt a database using the incorrect passphrase — whether an attempt by outsiders to use the app, or the user "fat-fingering" the passphrase and making a typo — you will get an exception:

```
11-19 09:17:22.700: E/SQLiteOpenHelper(1634):
net.sqlcipher.database.SQLiteException: file is encrypted or is not a
database
```

Alas, this is not a specific exception, making it a bit difficult to detect failed passphrases specifically. Your options are:

- Assume that your testing is sound and that exceptions when opening a database represent invalid passphrases, or
- Use a generic error message that hints at an invalid passphrase but leaves open the possibility of something else being wrong, or
- Read into the exception's message looking for "file is encrypted or is not a database", though this is fragile in the face of changes to SQLCipher for Android

SQLCipher for Android and Performance

Some developers worry about the overhead that encryption will place on the database I/O, and therefore worry that SQLCipher for Android will make their app unacceptably slow.

The impact of SQLCipher is not that bad, particularly for hardware with faster CPUs. Encryption is CPU-intensive, so faster CPUs reduce the overhead of the encryption. Also, since the disk I/O is comparable between SQLite and SQLCipher, the fact that flash memory is slow will mean that disk I/O, not decryption speed, will be the primary determinant of the speed of your queries. Similarly, disk I/O will count for more than CPU speed for the encryption needed for INSERT/UPDATE/DELETE operations.

For example, porting one relatively crude benchmark to use SQLCipher for Android showed no statistically significant performance difference from the SQLite edition on a Nexus 5 running Android 4.4.2.

To the extent that encryption adds overhead, it will tend to magnify existing problems. For example, anything that involves a "table scan" (i.e., a non-indexed lookup of database contents) will need more pages to be decrypted and, therefore, more decryption time. If your database I/O is well-tuned for SQLite, such as adding appropriate indexes, then your SQLCipher for Android overhead should be nominal.

Of course, the worse the CPU, the worse the story, and so older/cheaper devices may fare worse with SQLCipher for Android by comparison.

Encrypted Preferences

There are effectively three forms of data storage in Android:

- SQLite databases
- `SharedPreferences`
- Arbitrary files, in whatever format you want

You can encrypt SQLite via SQLCipher for Android, as seen in this chapter. You can encrypt arbitrary files as part of your data format, such as via `javax.crypto`.

What is not supported, out of the box, is a way to encrypt `SharedPreferences`.

There are two approaches for encrypting the contents of `SharedPreferences`:

1. Encrypt the container in which the `SharedPreferences` are stored
2. Encrypt each preference value as you store it in the `SharedPreferences`, and decrypt it when you read the value back out

Encryption via Custom SharedPreferences

`SharedPreferences` is an interface. Hence, you can create other implementations of that interface that store their data in something other than unencrypted XML files.

`CWSharedPreferences` is one such implementation. You can find it in the `cwac-prefs` project on GitHub.

`CWSharedPreferences` handles the `SharedPreferences` and `SharedPreferences.Editor` interfaces, along with the in-memory representations of the preferences. It then delegates the work of *storing* the preferences to a strategy object, implementing a strategy interface (`CWSharedPreferences.StorageStrategy`). Two such strategy implementations are supplied in the project: one using ordinary SQLite, and one using SQLCipher for Android.

The basic recipe for using `CWSharedPreferences` is:

- Create the strategy object, such as

```
new SQLCipherStrategy(getContext(), NAME, "atestpassword", LoadPolicy.SYNC)
```

(here, `NAME` is the name of the set of preferences, `"atestpassword"` is your passphrase, and `LoadPolicy.SYNC` indicates that the preferences should be loaded from disk immediately, not on a background thread)

- Create a `CWSharedPreferences` that employs your chosen strategy:

```
new CWSharedPreferences(yourStrategyObjectGoesHere);
```

- Use the `CWSharedPreferences` as you would any other `SharedPreferences` implementation
- Call `close()` on the strategy object, to release any resources that it might hold (e.g., open database connection)

Encryption via Custom Preference UI and Accessors

The big drawback to the custom `SharedPreferences` is the fact that you cannot get the `PreferenceScreen` system to work with it. The preference UI is hard-wired to use the stock implementation of `SharedPreferences` and does not appear to support any way to substitute in some other implementation.

Hence, another approach is to keep things in standard `SharedPreferences`' XML files, but encrypt text values on a preference-by-preference basis. Since the data type needs to remain the same, most likely you would restrict this to encrypting strings (e.g., `EditTextPreference`, `ListPreference`) rather than numbers, booleans, etc.

To do this, you would need to:

- Implement static methods somewhere for your encryption and decryption algorithms
- Subclass the `Preference` classes of interest and override methods that would deal with the raw preference data, like `onDialogClosed()`, to encrypt the values you persist and decrypt the values you read in, using the static methods mentioned above
- Use your extended `Preference` classes in your preference XML as needed
- Use those static methods as part of reading (or writing) the preference values directly via `SharedPreferences`

The downsides to this approach include:

- Only certain preferences are encrypted, rather than all of them
- You lose some of the low-level encryption power of SQLCipher for Android, such as automatic hashing of passphrases, which you would have to handle yourself
- There may not be a library that supplies these extended `Preference` classes, forcing you to roll your own

IOCipher

SQLCipher for Android is also used as the backing store for IOCipher. IOCipher is a virtual file system (VFS) for Android, allowing you to write code that looks and works like it uses normal file I/O, yet all of the files are actually saved as BLOBs in a SQLCipher for Android database. The result is a fully-encrypted VFS, inheriting all of SQLCipher's security features, such as default AES-256 encryption. This may be easier for you to use than encrypting and decrypting files individually via `javax.crypto`, for example.

IOCipher is considered to be in pre-alpha state as of November 2012.

Packaging and Distributing Data

Sometimes, you not only want to ship your code and simple resources with your app, but you also want to ship other types of data, such as an initial database that your app will use when first run. This chapter will examine the means by which you can do those sorts of things.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- database access
- content provider theory
- content provider implementations

Packing a Database To Go

Android's support for databases is focused on databases you create and populate entirely at runtime. Even if you want some initial data in the database, the expectation is that you would add that via Java code, such as the series of `insert()` calls we made in the `DatabaseHelper` of the various flavors of the `ConstantsBrowser` sample application.

However, that is tedious and slow for larger initial data sets, even if you make careful use of transactions to minimize the disk I/O.

What would be nice is to be able to ship a pre-populated database with your app. While Android does not offer built-in support for this, there are a few ways you can accomplish it yourself. One of the easiest, though, is to use existing third-party code

that supports this pattern, such as Jeff Gilfelt's `SQLiteAssetHelper`, available via a GitHub repository.

Android Studio users can add a `implementation` statement to the `dependencies` closure in `build.gradle` to pull in `com.readystatesoftware.sqliteasset:sqliteassethelper:...` (for some version indicated by `...`).

`SQLiteAssetHelper` replaces your existing `SQLiteOpenHelper` subclass with one that handles database creation and upgrading for you. Rather than you writing a lot of SQL code for each of those, you provide a pre-populated SQLite database (for creation) and a series of SQL scripts (for upgrades). `SQLiteAssetHelper` then does the work to set up your pre-populated database when the database is first accessed and running your SQL scripts as needed to handle schema changes. And, `SQLiteAssetHelper` is open source, licensed under the same Apache License 2.0 that is used for Android proper.

To examine `SQLiteAssetHelper` in action, let's look at the `Database/ConstantsAssets-AndroidStudio` sample project. This is yet another rendition of the same app as the other flavors of `ConstantsBrowser`, but one where we use a pre-populated database.

Create and Pack the Database

Whereas normally you create your SQLite database at runtime from Java code in your app, you now create your SQLite database using whatever tools you like, at development time. Whether you use the command-line `sqlite3` utility, the SQLite Manager extension for Firefox, or anything else, is up to you. You will need to set up all of your tables, indexes, and so forth.

Then, you need to:

1. Create an `assets/databases/` directory in your project
2. Copy your database into this directory (or put it there in the first place, if you prefer)

If your `minSdkVersion` is less than 11, you will instead need to have a ZIP or GZIP archive containing the database. The archive should have the same name as the database file, just with the `.zip` or `.gz` extension. The reason for the ZIP compression comes from an Android 1.x/2.x limitation – assets that are compressed by the Android build tools have a file-size limitation (around 1MB). Hence, you need

to store larger files in a file format that will not be compressed by the Android build tools, and those tools will not try to compress a `.zip` file.

In the `ConstantsAssets` project, you will see an `assets/databases/constants.db` file, containing a copy of the SQLite database with our `constants` table and pre-populated values.

Unpack the Database, With a Little Help(er)

Your compressed database will ship with your APK. To get it into its regular position on internal storage, you use `SQLiteAssetHelper`. Simply create a subclass of `SQLiteAssetHelper` and override its constructor, supplying the same values as you would for a `SQLiteOpenHelper` subclass, notably the database name and schema revision number. Note that the database name that you use must match the filename of the compressed database (minus the `.zip` extension, if you needed that).

So, for example, our new `DatabaseHelper` looks like this:

```
package com.commonsware.android.dbasset;

import android.content.Context;
import com.readystatesoftware.sqliteasset.SQLiteAssetHelper;

class DatabaseHelper extends SQLiteAssetHelper {
  static final String TITLE="title";
  static final String VALUE="value";
  static final String TABLE="constants";
  private static final String DATABASE_NAME="constants.db";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }
}
```

(from Database/ConstantsAssets-AndroidStudio/app/src/main/java/com/commonsware/android/dbasset/DatabaseHelper.java)

`SQLiteAssetHelper` will then copy your database out of assets and set it up for conventional use, as soon as you call `getReadableDatabase()` or `getWritableDatabase()` on an instance of your `SQLiteAssetHelper` subclass.

Upgrading Sans Java

Traditionally, with `SQLiteOpenHelper`, to handle a revision in your schema, you override `onUpgrade()` and do the upgrade work in there. With `SQLiteAssetHelper`, there is a built-in `onUpgrade()` method that uses SQL scripts in your APK to do the upgrade work instead.

These scripts will also reside in your `assets/databases/` directory of your project. The name of the file will be `$NAME_upgrade_$FROM-$TO.sql`, where you replace `$NAME` with the name of your database (e.g., `constants.db`), `$FROM` with the old schema version number (e.g., `1`) and `$TO` with the new schema version number (e.g., `2`). Hence, you wind up with files like `assets/databases/constants.db_upgrade_1-2.sql`. This should contain the SQL statements necessary to upgrade your schema between the versions.

`SQLiteAssetHelper` will chain these together as needed. Hence, to upgrade from schema version 1 to 3, you could either have a single dedicated 1->3 script, or a 1->2 script and a 2->3 script.

Limitations

The biggest limitation comes with disk space. Since APK files are read-only at runtime, you cannot delete the copy of the database held as an asset in your APK file once `SQLiteAssetHelper` has unpacked it. This means that the space taken up by your ZIP file will be taken up indefinitely. Note, though, that you could use this to your advantage, offering the user a "start over from scratch" option that deletes their existing database, so `SQLiteAssetHelper` will unpack a fresh original copy on the next run. Or, you could implement a `SQLiteDownloadHelper` that follows the `SQLiteAssetHelper` approach but obtains its database from the Internet instead of from assets.

In principle, SQLite could change their file format. If that ever happens, you will need to make sure that you create a SQLite database in the file format that can be used by Android, more so than what can be used by the latest SQLite standalone tools.

Advanced Database Techniques

This chapter offers tips and techniques for working with SQLite beyond what the previous chapters in the book have covered.

Prerequisites

This chapter assumes that you have read the core chapters, particularly the ones on databases and Internet access.

Also, please read the chapter on advanced action bar techniques, particularly the section on `SearchView`, as that is used in one of the sample apps.

Full-Text Indexing

Standard SQL databases are great for ordinary queries. In particular, when it comes to text, SQL databases are great for finding rows where a certain column value matches a particular string. They are usually pretty good about finding when a column value matches a particular string prefix, if there is an index on that column. Things start to break down when you want to search for an occurrence of a string in a column, as this usually requires a "table scan" (i.e., iteratively examining each row to see if this matches). And getting more complex than that is often impossible, or at least rather difficult.

SQLite, in its stock form, inherits all those capabilities and limitations. However, SQLite also offers full-text indexing, where we can search our database much like how we use a search engine (e.g., "find all rows where this column has both `foo` and `bar` in it somewhere"). While a full-text index takes up additional disk space, the speed of the full-text searching is quite impressive.

For example, if you are reading this book using the Android APK edition (instead of the PDF, EPUB, or Kindle/MOBI editions), tap on the `SearchView` action bar item and search for `FTS4`. You will get a list of matches back almost instantaneously, despite the fact that you are searching a multi-megabyte book. That is because this book ships a SQLite-powered full-text index of the book's contents, specifically to power your use of `SearchView`.

In this section, we will review how you can add full-text indexing to your SQLite database and how you can let the user take advantage of that index using a `SearchView`.

First, a Word About SQLite Versions

SQLite has evolved since Android's initial production release in 2008.

In many cases, Android does not incorporate updates to third-party code, for backwards-compatibility reasons (e.g., Apache's HttpClient). In the case of SQLite, newer Android versions do take on newer versions of SQLite... but the exact version of SQLite that a given version of Android uses is undocumented. Worse, some device manufacturers replace the stock SQLite for a version of Android with a different one.

This Stack Overflow answer contains a mapping of Android OS releases to SQLite versions, including various "anomalies" where manufacturers have elected to ship something else.

In many cases, the SQLite version does not matter. Core SQLite capabilities will have existed since the earliest days of Android. However, full-text indexing did not exist in the first SQLite used by Android, meaning that you will have to pay attention to your `minSdkVersion` and aim high enough that devices should support the full-text indexing option you choose.

Note that you could use an external SQLite implementation, one that gives you a newer SQLite engine than what might be on the device. For example, SQLCipher for Android ships its own copy of SQLite (with the SQLCipher extensions compiled in), one that is often newer than the one that is baked into the firmware of any given device.

FTS3 and FTS4

There are two full-text indexing options available in SQLite: FTS3 and FTS4. FTS4 can be much faster on certain queries, though overall the speed of the two implementations should be similar. FTS4 has two key limitations:

1. It may take a bit more disk space for its indexes.
2. It was added to SQLite 3.7.4, which was only introduced into standard Android in API Level 11.

The sample app for this section will demonstrate FTS4, as that is available on most Android devices.

Note that the Android developer documentation does not cover FTS3 or FTS4 full-text indexing. The details for the SQL syntax to support these options can be found in the SQLite documentation.

Creating a Full-Text Indexed Table

A full-text indexed table, using FTS3 or FTS4, uses SQLite's `CREATE VIRTUAL TABLE` syntax. This indicates that you are opting into some special table-storage behavior, rather than the stock stuff.

In the `Database/FTS` sample project, the `onCreate()` method of our `SQLiteOpenHelper` subclass (`DatabaseHelper`) creates such a virtual table, using FTS4 for full-text indexing:

```
@Override
public void onCreate(SQLiteDatabase db) {
  db.execSQL("CREATE VIRTUAL TABLE questions USING fts4("
                +"_id INTEGER PRIMARY KEY, title TEXT, "
                +"link TEXT, profileImage TEXT, creationDate INTEGER, "
                +"order=DESC);");
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/DatabaseHelper.java)

There are a few differences here from a typical `CREATE TABLE` statement, beyond the introduction of the `VIRTUAL` keyword:

- The `USING fts4` indicates that the virtual table is employing the FTS4 full-text indexing engine. To use FTS3, just replace `fts4` with `fts3`.

- You can have key-value pairs in the column list, separated by equals signs, to provide options for configuring the virtual table. In this case, it will provide options for configuring the FTS4 indexing behavior. In this case, we are providing `order=DESC`, to indicate that the full-text index should be optimized for returning items in descending order. Note that these options only exist for FTS4, not FTS3. The full roster of available options is covered in the SQLite documentation.

This gives us a table that supports normal table operations but also has a full-text index for its columns. However, there are some limitations, notably that these tables ignore constraints. So, for example, the `PRIMARY KEY` constraint applied to the `_id` column is ignored.

Populating a Full-Text Indexed Table

Adding content to an FTS3 or FTS4 table uses the same `INSERT` statements that you might use for a regular table. For example, the `DatabaseHelper` in the sample app has an `insertQuestions()` method that deletes all existing rows in the `questions` table, then inserts a bunch of rows based on a supplied `List` of `Item` objects:

```
void insertQuestions(Context app, List<Item> items) {
  SQLiteDatabase db=getDb(app);

  db.beginTransaction();

  db.delete("questions", null, null);

  try {
    for (Item item : items) {
      Object[] args={ item.id, item.title, item.link,
                      item.owner.profileImage, item.creationDate};

      db.execSQL("INSERT INTO questions (_id, title, "
                    +"link, profileImage, creationDate) "
                    +"VALUES (?, ?, ?, ?, ?)",
                  args);
    }

    db.setTransactionSuccessful();
  }
  finally {
    db.endTransaction();
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/DatabaseHelper.java)

If those `Item` objects look familiar, that is because this app is a modified version of the Stack Overflow questions apps profiled in the chapter on Internet access.

The reason why we are deleting everything before inserting is just to keep the sample simple. The database table will hold all of the questions pulled from the Stack Exchange API. Each time we run the app, we get the latest questions from that API. The vision was to use `INSERT OR REPLACE` or `INSERT OR IGNORE` statements to be able to merge content into the table. However, FTS3 and FTS4 tables ignore all constraints, as noted above, which prevents the conflict resolution options (e.g., `OR REPLACE`) from working. Hence, rather than manually sifting through to find if there is an existing row or not for a given ID value, this sample simply gets rid of all existing rows. A production-grade app would likely apply a more sophisticated algorithm.

Querying a Full-Text Indexed Table

While you can query a full-text indexed table using normal `SELECT` statements, usually the point is to apply the `MATCH` operator, as is seen in the `loadQuestions()` method from `DatabaseHelper`:

```
Cursor loadQuestions(Context app, String match) {
  SQLiteDatabase db=getDb(app);

  if (TextUtils.isEmpty(match)) {
    return(db.rawQuery("SELECT * FROM questions ORDER BY creationDate DESC",
                       null));
  }

  String[] args={ match };

  return(db.rawQuery("SELECT * FROM questions WHERE title "
                     +"MATCH ? ORDER BY creationDate DESC", args));
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/DatabaseHelper.java)

The `MATCH` operator supports a wide range of query structures, including:

- Keyword matches (e.g., `Android`)
- Prefix matches (e.g., `SQL*`)
- Phrase matches (e.g., `"open source"`)
- `NEAR`, `AND`, `OR`, and `NOT` operators (e.g., `sqlite AND database`)

The result is the same sort of `Cursor` that you would get from a regular `SELECT` statement against a non-full-text-indexed table.

Some Notes About the Rest of the Sample App

As noted previously, this sample app is a revised version of the Stack Overflow questions list from the chapter on Internet access. It is specifically derived from the Picasso version of the sample. However, this version is designed to allow the user to full-text search the downloaded question data (e.g., title), above and beyond just seeing the list of latest questions.

This, in turn, requires a few more changes than those outlined so far. The following sections outline some of the highlights.

Adding a ModelFragment

The original sample had a very simple data model: a list of questions retrieved via Retrofit. Hence, the sample did not include much in the way of model management.

The FTS sample needs a database, which implies more local disk I/O that we are responsible for, which in turn leads us in the direction of implementing a model fragment (`ModelFragment`), much as the tutorials and a few other samples do:

```
package com.commonsware.android.fts;

import android.app.Activity;
import android.app.Fragment;
import android.content.Context;
import android.database.Cursor;
import android.os.Bundle;
import android.util.Log;
import org.greenrobot.eventbus.EventBus;
import org.greenrobot.eventbus.Subscribe;
import org.greenrobot.eventbus.ThreadMode;
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;

public class ModelFragment extends Fragment {
  private Context app=null;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setRetainInstance(true);
  }

  @Override
  public void onAttach(Activity host) {
```

```
  super.onAttach(host);

  EventBus.getDefault().register(this);

  if (app==null) {
    app=host.getApplicationContext();
    new FetchQuestionsThread().start();
  }
}

@Override
public void onDetach() {
  EventBus.getDefault().unregister(this);

  super.onDetach();
}

@Subscribe(threadMode =ThreadMode.BACKGROUND)
public void onSearchRequested(SearchRequestedEvent event) {
  try {
    Cursor results=DatabaseHelper.getInstance(app).loadQuestions(app, event.match);

    EventBus.getDefault().postSticky(new ModelLoadedEvent(results));
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
        "Exception searching database", e);
  }
}

class FetchQuestionsThread extends Thread {
  @Override
  public void run() {
    Retrofit retrofit=
      new Retrofit.Builder()
        .baseUrl("https://api.stackexchange.com")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
    StackOverflowInterface so=
      retrofit.create(StackOverflowInterface.class);

    try {
      SOQuestions questions=so.questions("android").execute().body();

      DatabaseHelper
          .getInstance(app)
          .insertQuestions(app, questions.items);
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
          "Exception populating database", e);
    }

    try {
      Cursor results=DatabaseHelper.getInstance(app).loadQuestions(app, null);

      EventBus.getDefault().postSticky(new ModelLoadedEvent(results));
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
```

```
            "Exception populating database", e);
      }
    }
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/ModelFragment.java)

In `onCreate()`, we mark this fragment as retained, as that is key to the model fragment pattern, so the fragment retains the model data across configuration changes.

In `onAttach()`, we register for the greenrobot EventBus, plus kick off a `FetchQuestionsThread` if we have not done so already (i.e., this is the first `onAttach()` call we have received). `onDetach()` unregisters us from the event bus.

`FetchQuestionsThread`, in turn, uses Retrofit to download the questions from Stack Overflow, then uses `DatabaseHelper` to insert the questions into the FTS-enabled database table, then uses the `DatabaseHelper` again to retrieve all existing questions in the form of a `Cursor`, which it wraps in a `ModelLoadedEvent` and posts to the `EventBus`. This time, though, it posts it as a sticky event.

That sticky event is consumed by a revised version of the `QuestionsFragment`, in its `onModelLoaded()` method:

```
@Subscribe(sticky = true, threadMode =ThreadMode.MAIN)
public void onModelLoaded(ModelLoadedEvent event) {
  ((SimpleCursorAdapter)getListAdapter()).changeCursor(event.model);

  if (sv!=null) {
    sv.setEnabled(true);
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

But because this is a sticky event, we will get this event both when it is raised (because the data is loaded) and any time thereafter when the fragment registers with the `EventBus`. This allows `QuestionsFragment` to *not* be retained, as it will get back the bulk of its model data automatically from greenrobot's EventBus.

`QuestionsFragment` also is modified from the Picasso sample to deal with the fact that its model data is now a `Cursor`, so it uses `SimpleCursorAdapter` to populate the list. To handle loading avatar images from the URLs, `QuestionsFragment` adds a

QuestionBinder implementation of ViewBinder to the SimpleCursorAdapter, where QuestionBinder handles the Picasso logic from before:

```
private class QuestionBinder implements SimpleCursorAdapter.ViewBinder {
  int size;

  QuestionBinder() {
    size=getActivity()
            .getResources()
            .getDimensionPixelSize(R.dimen.icon);
  }

  @Override
  public boolean setViewValue (View view, Cursor cursor, int columnIndex) {
    switch (view.getId()) {
      case R.id.title:
        ((TextView)view).setText(Html.fromHtml(cursor.getString(columnIndex)));

        return(true);

      case R.id.icon:
        Picasso.with(getActivity()).load(cursor.getString(columnIndex))
            .resize(size, size).centerCrop()
            .placeholder(R.drawable.owner_placeholder)
            .error(R.drawable.owner_error).into((ImageView)view);

        return(true);
    }

    return(false);
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

The main activity (MainActivity) sets up the ModelFragment in onCreate(), at least when one does not already exist due to a configuration change:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
    getFragmentManager().beginTransaction()
                        .add(android.R.id.content,
                             new QuestionsFragment()).commit();
  }

  model=(ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

  if (model==null) {
    model=new ModelFragment();
    getFragmentManager().beginTransaction().add(model, MODEL).commit();
```

```
    }
  }
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/MainActivity.java)

This description, though, has skipped over the `onEventBackgroundThread()` method on the `ModelFragment`, which we will get to later in this overview.

Adding a SearchView

As is covered in the chapter on advanced action bar techniques, a `SearchView` can be used to provide the standard "magnifying glass" search icon in the action bar. When tapped, the action bar item expands into a field where the user can type something, which our code can then receive and use to update the UI. In the `SearchView` sample from the action bar chapter, we saw using a `SearchView` for filtering. This time, we will use a `SearchView` for searching.

For a search, we need to know when the user is done typing, which is usually done by the user clicking a submit button. Hence, our code to configure the `SearchView` (a `configureSearchView()` method in `QuestionsFragment`) calls `setSubmitButtonEnabled(true)`:

```
  private void configureSearchView(Menu menu) {
    MenuItem search=menu.findItem(R.id.search);

    search.setOnActionExpandListener(this);
    sv=(SearchView)search.getActionView();
    sv.setOnQueryTextListener(this);
    sv.setSubmitButtonEnabled(true);
    sv.setIconifiedByDefault(true);

    if (initialQuery != null) {
      sv.setIconified(false);
      search.expandActionView();
      sv.setQuery(initialQuery, true);
    }
  }
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

This, in turn, means that we need to pay attention to `onQueryTextSubmit()` in our `SearchView.OnQueryTextListener` implementation. That interface is implemented on `QuestionsFragment` itself, and delegates its work to a `doSearch()` method:

```
@Override
public boolean onQueryTextSubmit(String query) {
  doSearch(query);

  return(true);
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

That method, in turn, confirms that the search is different than the last one we did (so we do not waste time running the search again), disables the `SearchView`, and posts a `SearchRequestedEvent` on the `EventBus` with the user's search string:

```
private void doSearch(String match) {
  if (!match.equals(lastQuery)) {
    lastQuery=match;

    if (sv != null) {
      sv.setEnabled(false);
    }

    EventBus.getDefault().post(new SearchRequestedEvent(match));
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

That event is picked up by `onSearchRequested()` on `ModelFragment`. The `@Subscribe(threadMode =ThreadMode.BACKGROUND)` annotation means that the event will be delivered to us on an EventBus-supplied background thread, so we can perform database I/O. In it, we call `loadQuestions()` on the `DatabaseHelper` to perform the search, and post another sticky `ModelLoadedEvent` to update the UI with the search results and re-enable the `SearchView`:

```
@Subscribe(threadMode =ThreadMode.BACKGROUND)
public void onSearchRequested(SearchRequestedEvent event) {
  try {
    Cursor results=DatabaseHelper.getInstance(app).loadQuestions(app, event.match);

    EventBus.getDefault().postSticky(new ModelLoadedEvent(results));
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
        "Exception searching database", e);
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/ModelFragment.java)

When the user clears the `SearchView`, such as by pressing the BACK button a few times, the `onMenuItemActionCollapse()` method of `QuestionsFragment` calls a `clearSearch()` method:

```
@Override
public boolean onMenuItemActionCollapse(MenuItem item) {
  clearSearch();

  return(true);
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

That `clearSearch()` method simply posts another `SearchRequestedEvent`, this time to load a fresh roster of all questions:

```
private void clearSearch() {
  if (lastQuery!=null) {
    lastQuery=null;

    sv.setEnabled(false);
    EventBus.getDefault().post(new SearchRequestedEvent(null));
  }
}
```

(from Database/FTS/app/src/main/java/com/commonsware/android/fts/QuestionsFragment.java)

The Results

When you run the app, you are initially presented with the list of questions pulled from the Stack Exchange API:

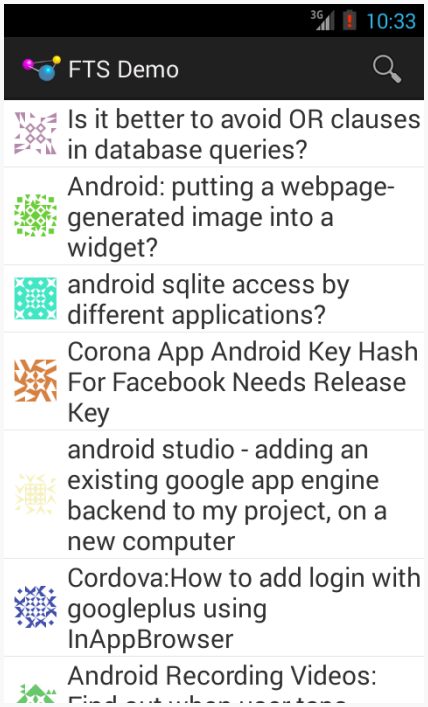


Figure 799: FTS Demo, As Initially Launched

Tapping on the `SearchView` opens it up, as normal, though this time with the "submit" button (the rightward-pointing arrowhead):

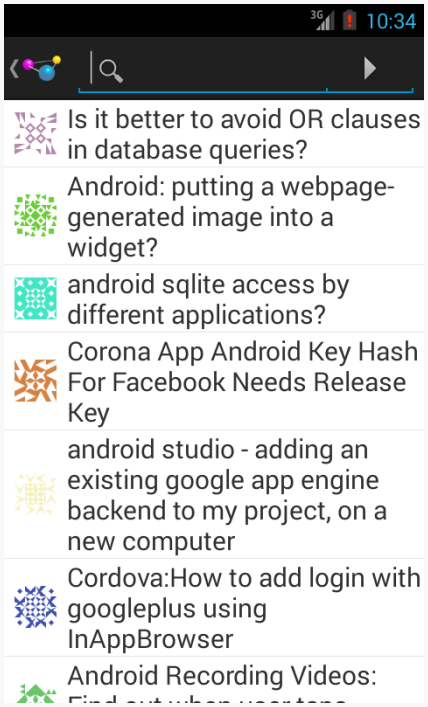


Figure 800: FTS Demo, with Open SearchView

Typing in a search, then tapping the "submit" button, will reload the list with those questions that match the search criteria in the question title:

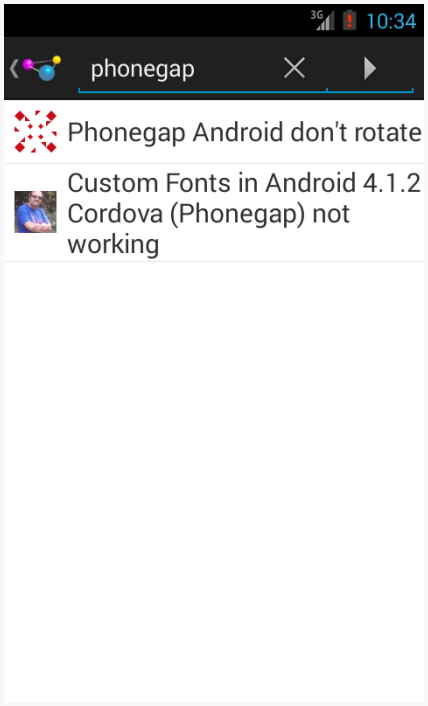


Figure 801: FTS Demo, Showing Basic Search

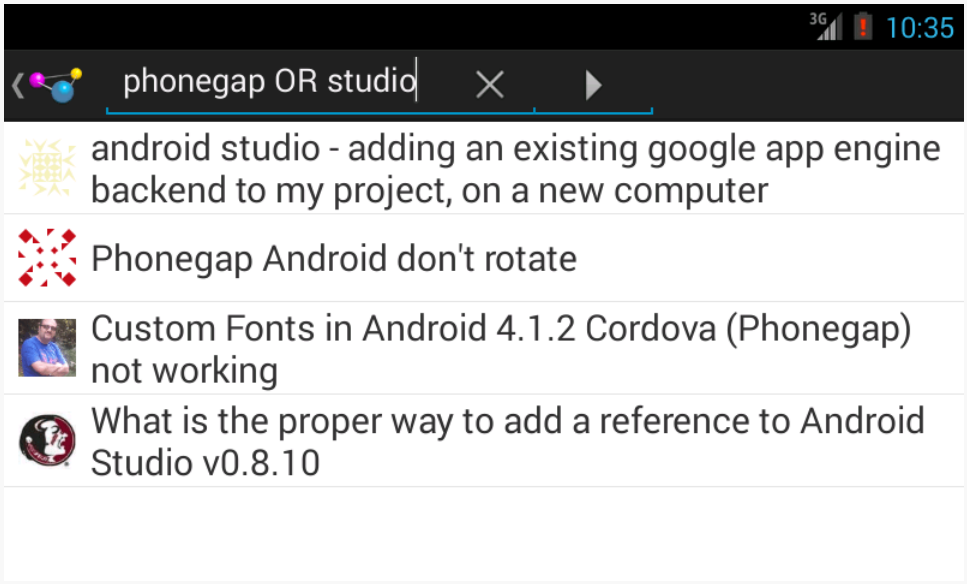


Figure 802: FTS Demo, Showing Boolean Search

Using the BACK button to get out of the `SearchView` reloads the full list of questions.

Getting Snippets

Usually, the content that is being indexed is a lot longer than Stack Overflow question titles. For example, it might be chapters in a book on Android application development. In that case, it would be useful to not only find out what chapters match the search expression, but what the prose is around the search expression, to help the user determine which search results are likely to be useful.

The APK edition of this book stores each paragraph and bullet as a separate entry in a SQLite database in an FTS3-enabled table. The query used when the reader types in a search expression in the app's `SearchView` is:

```
SELECT ROWID as _id, file, node, snippet(booksearch) AS snippet FROM
booksearch WHERE prose MATCH ?
```

Here, `file` and `node` are used to identify where this passage came from within the book, so when the user taps on a search result in the list, the book reader can jump to that particular location.

The `snippet()` auxiliary function will return, as the name suggests, a snippet of the indexed text, with the search match highlighted. It takes the name of the table `booksearch` as a mandatory parameter. It also supports optional parameters for what to bracket the search match with (defaults to `<b>` and `</b>`) and what to use for an ellipsis for extended prose segments (defaults to `<b>...</b>`). In the case of this query, the default formatting of the result is used. The resulting text can then be fed into `Html.fromHtml()` to generate the text for the `ListView` row, showing the search match within the snippet highlighted in bold:

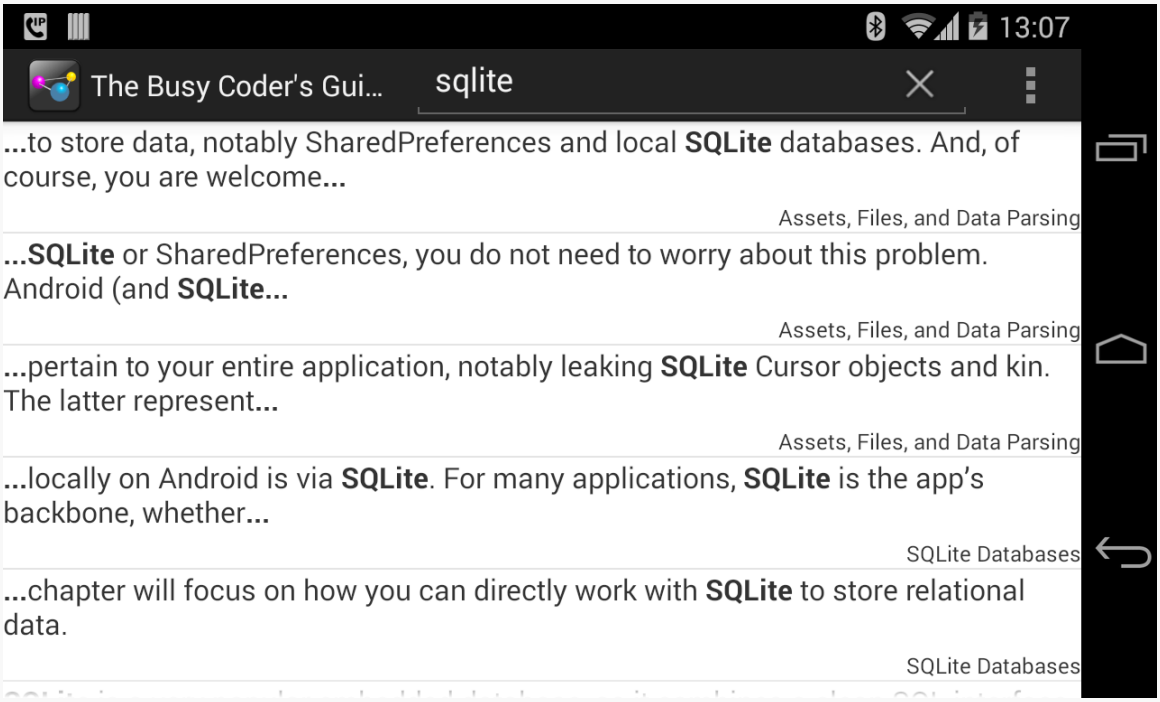


Figure 803: This Book's Reader App, Showing Search Results

The app also shows the name of the chapter in the lower-right corner of each row, to help provide larger context for where this snippet comes from.

Data Backup

Backing up your PC used to be essential. To some extent, it still is, but as more and more stuff moves to "the cloud", local machine backups become less and less important.

Backing up mobile devices historically has been an afterthought, as a lot of what people use these devices for are gateways to Internet-hosted content and services. However, as more and more stuff becomes local to the device — for disconnected operation, for example — the greater the need for backing up that local data.

Android does not have a full-device backup as part of the OS. It does have some hooks that Google advertises as being "backup", but IT professionals would not consider Google's definition to match their own for "backup". And, what hooks there are exist at the level of an app, not the device, providing opportunity — and requirements — for developers to tailor what gets backed up and, to a lesser extent, how it gets backed up.

This chapter will explore the steps to back up your app's data, with and without Google's assistance.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on file access and Internet access.

Having read the chapters on SSL and SQLCipher for Android are not required but may prove to be useful background for some of the side topics in this chapter.

First, Some Terminology

One key concept when it comes to backups is what, exactly, we are backing up. The general rule is that you focus your backup regimen on the "system of record". This is the one and only system that has the master copy of the data. While it may be one "system", that "system" may be rather complex (e.g., cluster of database servers). However, anything else outside of that system — such as clients for those servers — are not part of the system of record. While they may have some data that is also held by the system of record, that data is considered to be a cached local copy; the system of record has the "real" copy of the data.

Differing Definitions of "Backup"

The problem is that we toss around the term "backup" as though there is a universal canonical definition for that term. Hence, what Google will tell users is "backup" will not necessarily line up with what an IT department will consider "backup" to mean.

What Google Thinks "Backup" Means

Google's focus is on the cloud. Therefore, their focus is on apps using data resident in the cloud, with some servers forming the system of record. Google (presumably) does some sort of backup for their own systems of record, for their own Internet-based services, and Google assumes that other firms are doing the same.

The side-effect of this definition, though, is that Google does not view an app as having much in the way of local data that needs to be backed up. Cached data can always be reloaded from the system of record, after all. What Google expects needs to be backed up will be local preferences and perhaps authentication or authorization credentials for working with the system of record. This dataset is small and does not necessarily change all that often.

Because the dataset does not change that often, Google only really cares about restoring that data in case of a total device replacement. In other words, if your phone gets run over by a bakery truck, and you wind up replacing that phone with another Android phone, Google is interested in making sure that your old phone's apps get restored along with the old phone's last backup of the tiny dataset. After that, you are on your own. In particular, because the dataset does not change that often and does not have much in the way of critical data, Google is not concerned with allowing users to restore app data from backup for any reason *other* than

replacing the device outright. In other words, Google is only concerned with disaster recovery.

Google does not offer any configurability for where backups themselves are stored. Whatever Google backs up, Google stores where Google wants. Terms of service and related agreements give Google — at least in Google's eyes — the right to do pretty much anything they want with that data. While they will tout the fact that Android 6.0+ backups are stored in an encrypted fashion, they fail to note that Google — not the developer, not the user – holds the encryption keys. Thus, the security offered by this encryption is nominal, perhaps slowing down somebody who breaks into Google's network, but otherwise not preventing anyone from accessing the data.

Also, there is a 25MB data cap on the size of the backup, so if your app might have data in excess of that, you need to handle backups yourself.

Finally, the author of this book cannot get Google's backup system to work on production hardware, as will be explained a bit more later in this chapter.

What IT Thinks "Backup" Means

Apps may well be the system of record for the data that they work with. There is no requirement that all apps be front-ends for some server, any more than there is a requirement that all desktop OS apps be front-ends for some server. There may be plenty of business or technical reasons why an app will be the system of record for its data, either all of the time or in between specific sync operations with some central data store.

As a result, an IT department will recognize that apps need a much more robust backup and restoration service, one that takes into account conventional IT backup concepts.

Most IT-grade backup regimens have the notion of "backup aging". Rather than Google's approach of considering only one backup to be relevant, an IT department will maintain a series of backups (e.g., 14 days of nightly backups, plus 3 months of weekly backups, plus 5 years of monthly backups), to be able to handle data that might be lost, but where that loss is not detected for some time.

Most IT-grade backups regimens allow data to be restored, in part or completely, at any point, not just in case a device is stepped on by an elephant or otherwise destroyed. Disaster recovery is a *scenario* of a backup regimen, not the sole objective.

IT departments also tend to be very concerned about where their business data goes. The idea that the data should be available, unencrypted, to arbitrary third parties would be an anathema. Business data should be backed up on by IT-supplied technology on IT-supplied backup media, employing whatever security the IT department thinks is necessary.

Suffice it to say, Google's approach to "backup" does not align well with what an IT department will want.

What Your Legal Counsel Thinks "Backup" Means

Legal counsel, at some point, should be brought into the discussion of backups, as, for better or worse, there are legal risks involved in backups.

Particularly with Google-style, send-the-data-to-a-third-party backups, you need to ensure that this will not get you in legal trouble. From European Union privacy laws to HIPAA in the US, there are plenty of laws that prohibit the careless distribution of data.

Beyond that, legal counsel will be worried about "the Ashley Madison scenario". A firm's IT department will be responsible for ensuring that their servers are not hacked into. However, once you start passing data to third parties, now you are at risk of *those* servers getting hacked into. Legal counsel can advise you on what your legal exposure is, in terms of potential lawsuits from people whose data might get leaked by these sorts of attacks.

Implementing IT-Style Backup

So, if we want to add backup and restore capability to our app, what is needed? To explore that, we will use the `Backup/BackupClient` sample project as an illustration. This is a clone of a sample that originally appeared in the chapter on files. We have a three-tab `ViewPager`, with a large `EditText` widget in each tab. The three tabs differ in where they persist their data:

- `getFilesDir()`
- `getExternalFilesDir()`
- `DIRECTORY_DOCUMENTS` — the user's `Documents/` directory on API Level 19+ devices

This revised sample adds backup-and-restore functionality to this app.

The app also has one other change: it stores the most-recently-visited tab in `SharedPreferences`. To that end, `MainActivity` has a `PrefsLoadThread` static inner class that asynchronously loads the `SharedPreferences`, then delivers them via greenrobot's EventBus:

```
private static class PrefsLoadThread extends Thread {
  private final Context ctxt;

  PrefsLoadThread(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
  }

  @Override
  public void run() {
    SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(ctxt);
    PrefsLoadedEvent event=new PrefsLoadedEvent(prefs);

    EventBus.getDefault().post(event);
  }
}

private static class PrefsLoadedEvent {
  private final SharedPreferences prefs;

  PrefsLoadedEvent(SharedPreferences prefs) {
    this.prefs=prefs;
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/MainActivity.java)

`MainActivity` picks up this event in `onPrefsLoaded()`, an EventBus method that takes `PrefsLoadedEvent` as a parameter and updates the current page of the `ViewPager` (named `pager`):

```
@Subscribe(threadMode =ThreadMode.MAIN)
public void onPrefsLoaded(PrefsLoadedEvent event) {
  this.prefs=event.prefs;

  int lastVisited=prefs.getInt(PREF_LAST_VISITED, -1);

  if (lastVisited>-1) {
    pager.setCurrentItem(lastVisited);
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/MainActivity.java)

The `PrefsLoadThread` is kicked off in `onStart()`, and the `PREF_LAST_VISITED` value is saved in `onStop()`, along with the registration and unregistration from the event bus:

```
@Override
protected void onStart() {
  super.onStart();

  EventBus.getDefault().register(this);

  if (prefs==null) {
    new PrefsLoadThread(this).start();
  }
}

@Override
protected void onStop() {
  EventBus.getDefault().unregister(this);

  if (prefs!=null) {
    prefs
      .edit()
      .putInt(PREF_LAST_VISITED, pager.getCurrentItem())
      .apply();
  }

  super.onStop();
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/MainActivity.java)

The net effect is that we retain the last-visited tab across invocations of `MainActivity`. This forms part of the data that we would like to back up.

Choosing the Backup Scope

The first question is: what exactly are we backing up? Files? Databases? `SharedPreferences`? Stuff that is out in common areas, like top-level directories on external storage (e.g., `DIRECTORY_DOCUMENTS`) or the `ContactsContract ContentProvider`?

Typically, an individual app will focus on backing up only that app's data, which would exclude the common areas from consideration. That does not mean that you

can't back up common data, but it makes restoration a bit more challenging, as you do not want to overwrite changes to that data that the user made from another app.

In `BackupClient`, we are backing up:

- the contents of `getFilesDir()`, which will hold onto one of our tabs' contents
- the contents of `getExternalFilesDir()`, which will hold onto another of our tabs' contents
- some of the contents of the directory that holds the `SharedPreferences` for the app, which will pick up the preference value we are using for the last-visited tab

Notably, we are not backing up the file out on shared storage (the "Public" tab, set to store its data in `Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS)`). Hence, whatever is in that tab will be left alone when we restore the data from the backup.

Choosing a Backup Trigger

The next question is: when are we backing up the data?

There are any number of possibilities:

- A push message, such as through GCM, could request that the app back up its data
- Time-based triggers, using `AlarmManager` or `JobScheduler`, could be used to periodically make backups
- You could offer backups on demand, such as through an action bar item

The automated options (push message, `AlarmManager`, `JobScheduler`) are great, so users do not forget to make a backup. On the other hand, there is the risk that the user is using the app at the time the automated backup is supposed to happen, which means you will need some additional logic to ensure that you postpone that backup until a quieter time. It is difficult to back up data that is actively in use.

The `BackupClient` sample will settle for a simple manual trigger, via a "Backup" action bar item in the main activity. We also have a "Restore" action bar item, to request to restore the data from a backup. So, `MainActivity` will load in a menu resource that contains these two options:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/backup"
    android:icon="@drawable/ic_backup_white_24dp"
    android:title="@string/menu_backup"/>
  <item
    android:id="@+id/restore"
    android:icon="@drawable/ic_restore_white_24dp"
    android:title="@string/menu_restore"/>
</menu>
```

(from Backup/BackupClient/app/src/main/res/menu/actions.xml)

It uses a pair of icons culled from Google's material design icon set.

That resource is inflated in `onCreateOptionsMenu()`. If the user chooses the "Backup" option, we start a `BackupService` to do the work:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.backup) {
      BackupService.enqueueWork(this);

      return(true);
    }
    else if (item.getItemId()==R.id.restore) {
      startActivity(new Intent(this, RestoreRosterActivity.class));

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/MainActivity.java)

We will get into the restore scenario a bit later in this chapter.

Generating the Dataset

Next, we need to actually collect the data to be backed up and package it in some form to send to a server to serve as the backup dataset.

There are any number of ways to package this sort of data, but a ZIP file seems like a likely candidate:

- It is fairly easy to work with on Android
- It is fairly easy to work with on servers that might need to unpack the data
- It is fairly easy to examine using desktop tools, for development, diagnostics, etc.

It is the job of the `BackupService` to create a ZIP file of our desired data, then send that ZIP file to a backup server.

`BackupService` itself is a `JobIntentService`, as this sort of work is a nice "fire-and-forget" sort of request, where we no longer need the service once the work is done.

In `onHandleWork()`, we orchestrate the major steps in this process:

```
@Override
public void onHandleWork(@NonNull Intent i) {
  try {
    File backup=buildBackup();

    uploadBackup(backup);
    backup.delete();

    EventBus.getDefault().post(new BackupCompletedEvent());
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(),
      "Exception creating ZIP file", e);
    EventBus.getDefault().post(new BackupFailedEvent());
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

We:

- Call a `buildBackup()` method that creates our backup dataset
- Call an `uploadBackup()` method to send the dataset to some backup server

- Delete the local backup when that is done, as we no longer need it
- Raise events on an event bus for the UI layer's use, for when a backup succeeds or fails

Those events can then trigger UI responses. In the case of this trivial sample app, they just result in `Toast` messages to the user:

```
@Subscribe(threadMode =ThreadMode.MAIN)
public void onCompleted(BackupService.BackupCompletedEvent event) {
  Toast
    .makeText(this, R.string.msg_backup_completed, Toast.LENGTH_LONG)
    .show();
}

@Subscribe(threadMode =ThreadMode.MAIN)
public void onFailed(BackupService.BackupFailedEvent event) {
  Toast
    .makeText(this, R.string.msg_backup_failed, Toast.LENGTH_LONG)
    .show();
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/MainActivity.java)

A production-grade app would do something more sophisticated, particularly for error messages, given that a `Toast` is ephemeral, and so the user might not see it.

`buildBackup()` is responsible for creating a file that contains our desired dataset and returning the `File` object pointing to that file:

```
private File buildBackup() throws IOException {
  File zipFile=new File(getCacheDir(), BACKUP_FILENAME);

  if (zipFile.exists()) {
    zipFile.delete();
  }

  FileOutputStream fos=new FileOutputStream(zipFile);
  ZipOutputStream zos=new ZipOutputStream(fos);

  zipDir(ZIP_PREFIX_FILES, getFilesDir(), zos);
  zipDir(ZIP_PREFIX_PREFS, getSharedPrefsDir(this), zos);
  zipDir(ZIP_PREFIX_EXTERNAL, getExternalFilesDir(null), zos);
  zos.flush();
  fos.getFD().sync();
  zos.close();
```

```
    return(zipFile);
  }
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

We put the backup ZIP file in internal storage cache (`getCacheDir()`), as that is not something that we are backing up, and therefore we do not need to worry about somehow trying to back up the backup file itself.

We then call `zipDir()` three times, one for each directory of data to be backed up. Two of the three locations have SDK-supplied methods to get the `File` object pointing at those directories: `getFilesDir()` and `getExternalFilesDir()`. Unfortunately, the SDK does not provide any direct method that returns a `File` pointing at the directory for `SharedPreferences`. So, we have to hack one ourselves, in the form of `getSharedPrefsDir()`:

```
  static File getSharedPrefsDir(Context ctxt) {
    return(new File(new File(ctxt.getApplicationInfo().dataDir),
      "shared_prefs"));
  }
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

`getApplicationInfo()` returns the `ApplicationInfo` object describing our app. That has a `dataDir` field that points to *all* of our internal storage (whereas `getFilesDir()` points to a subdirectory off of `dataDir`). The `SharedPreferences` are stored in XML files in a `shared_prefs/` directory off of the location pointed to by the `dataDir` field. This is not an ideal solution, as in theory the `SharedPreferences` storage location could move. However, this code should work for all API levels from 1 through 23, and therefore it is reasonably likely that it will hold up over time.

`zipDir()` not only takes the `File` of data to be backed up and a `ZipOutputStream` representing where to package the data, but it also takes a path prefix. ZIP files do not really have a directory structure; that structure is faked based on path-style names associated with each entry. The prefix is added to each of those names, giving the effect of putting each directory's contents into a separate "directory" within the ZIP archive. Those three prefixes are defined as simple `String` constants:

```
  static final String ZIP_PREFIX_FILES="files/";
  static final String ZIP_PREFIX_PREFS="shared_prefs/";
  static final String ZIP_PREFIX_EXTERNAL="external/";
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

`zipDir()` itself (mostly) is a typical recursive put-the-files-in-the-archive method:

```
private void zipDir(String basePath, File dir,
                    ZipOutputStream zos) throws IOException {
  byte[] buf=new byte[16384];

  if (dir.listFiles()!=null) {
    for (File file : dir.listFiles()) {
      if (file.isDirectory()) {
        String path=basePath+file.getName()+"/";

        zos.putNextEntry(new ZipEntry(path));
        zipDir(path, file, zos);
        zos.closeEntry();
      }
      else if (!file.getName().equals(BACKUP_PREFS_FILENAME)) {
        FileInputStream fin=new FileInputStream(file);
        int length;

        zos.putNextEntry(
          new ZipEntry(basePath+file.getName()));

        while ((length=fin.read(buf))>0) {
          zos.write(buf, 0, length);
        }

        zos.closeEntry();
        fin.close();
      }
    }
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

The one wrinkle is that we filter out files with a particular name, denoted by the `BACKUP_PREFS_FILENAME` constant:

```
private static final String BACKUP_PREFS_FILENAME=
  "com.commonsware.android.backup.BackupService.xml";
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

We will explore what this file is, and why we are not backing it up, later in this chapter.

This backup approach has its flaws, in the interests of keeping the example simple:

- The UI layer is not saving all in-flight data before doing the backup. Hence, any changes in the current tab, since we moved to that tab, are not saved to disk or backed up. And, since we only save what the current tab is in `onStop()`, that too has not been adjusted since our activity moved to the foreground, and so it may be out of date. A production-grade app will need to decide what data that has not been saved through ordinary means should be saved prior to a manual backup, assuming that the app has a manual backup option in the first place.
- The UI layer is not preventing the user from changing data that is being backed up while the backup is happening. In this sample app, the data to be backed up is small enough that it will probably happen quickly enough to not be a problem. A production-grade app, though, should take steps to prevent data entry (though perhaps not navigation through the app) while the backup is going on. Any such steps, though, need to take into account the possibility that the backup may fail — we do not want a failed backup to block the user from working in the app for hours.

Transmitting the Dataset

Given the data to be backed up in a nice convenient package, we need to get that dataset off the device and someplace safe, where we can later download and restore it if needed. There are any number of possible solutions here, including many existing public Web services (Dropbox, Amazon's AWS S3, Google Drive, etc.). If you are only worried about manual backups, you could even consider using `ACTION_SEND` to send the dataset as an email attachment, though size and content limitations on email attachments may make this impractical for many users.

`BackupService` works with some implementation of a particular REST-style API for backing up and restoring the data. This API is fairly lightweight, light enough that it can be implemented in ~70 lines of Ruby code, as will be seen later in this chapter. You could implement the same sort of API in any number of Web frameworks.

For backing up data, there are two REST operations that we need to perform:

- We need to create a new backup entry, via an HTTP `POST` request to `/api/backups` on the backup server
- We need to upload the dataset itself, via an HTTP `PUT` request to `/api/backups/.../dataset` on the backup server, where the `...` is a backup ID that we get from the response to the original `POST` request

To implement the client side, `BackupService` employs the OkHttp library profiled in the chapter on Internet access. Specifically, `uploadBackup()` does both of the HTTP requests necessary to back up the data, given the `File` pointing to the ZIP archive that is our dataset:

```
private void uploadBackup(File backup) throws IOException {
  Request request=new Request.Builder()
    .url(URL_CREATE_BACKUP)
    .post(RequestBody.create(JSON, "{}"))
    .build();
  Response response=OKHTTP_CLIENT.newCall(request).execute();

  if (response.code()==201) {
    String backupURL=response.header("Location");

    request=new Request.Builder()
      .url(backupURL+RESOURCE_DATASET)
      .put(RequestBody.create(ZIP, backup))
      .build();
    response=OKHTTP_CLIENT.newCall(request).execute();

    if (response.code()==201) {
      String datasetURL=response.header("Location");
      SharedPreferences prefs=
        getSharedPreferences(getClass().getName(),
          Context.MODE_PRIVATE);

      prefs
        .edit()
        .putString(PREF_LAST_BACKUP_DATASET, datasetURL)
        .commit();
    }
    else {
      Log.e(getClass().getSimpleName(),
        "Unsuccessful request to upload backup");
    }
  }
  else {
    Log.e(getClass().getSimpleName(),
      "Unsuccessful request to create backup");
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

We create an OkHttp `Request.Builder` representing our `POST` request. The URL is defined as a constant, `URL_CREATE_BACKUP`:

```
private static final String URL_CREATE_BACKUP=
  BuildConfig.URL_SERVER+"/api/backups";
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

This, in turn, is built up from the fixed REST endpoint path (`/api/backups`), with the rest of the URL coming from `BuildConfig.URL_SERVER`. This is defined out in our `build.gradle` file, allowing us to have different backup server locations based upon build types (or, in principle, product flavors):

```
buildTypes {
  debug {
      buildConfigField "String", "URL_SERVER", '"http://10.0.2.2:4567"'
  }

  release {
    buildConfigField "String", "URL_SERVER", '"http://10.0.2.2:4567"'
  }
}
```

(from Backup/BackupClient/app/build.gradle)

Here, they happen to both point to the same value at the moment, the IP address that, on an Android emulator, represents `localhost` of your development machine. However, you could easily change the `release` build type to point to some production instance of a backup server.

The body of the `POST` request is a JSON object containing whatever we want, in case we need to provide some sort of identifiers with the backup for server-side use or analysis. In this case, we are passing an empty JSON object (`{}`), using the `JSON` `MediaType` declared as another constant:

```
private static final MediaType JSON=
  MediaType.parse("application/json; charset=utf-8");
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

We then use an instance of an `OkHttpClient` object to perform the request, getting the `Response` synchronously (since we are already on a background thread). If multiple components in your app will all be using OkHttp, the recommendation is to use a singleton instance of `OkHttpClient`, here defined on `BackupService` itself:

```
static final OkHttpClient OKHTTP_CLIENT=new OkHttpClient();
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

The REST protocol to the backup server is that a 201 response code ("Created") means that our backup metadata has been saved and an ID has been generated for our backup. The `Location` header in the response contains a REST URL pointing to the backup itself (`/api/backups/...` for some value of `...`). We then use that to generate the URL for the dataset (`/api/backups/.../dataset`), and perform a `PUT` request for the dataset, using the `ZIP` `MediaType` defined as yet another constant:

```
  private static final MediaType ZIP=
    MediaType.parse("application/zip");
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupService.java)

Once again, a 201 response indicates that our resource was created, and the `Location` header provides the URL for the backup dataset. We stuff that URL in a `SharedPreferences` object unique to `BackupService`, under a `PREF_LAST_BACKUP_DATASET` key. We will use that — at least, in theory – if we are restored from a Google disaster recovery process. We will explore that more later in the chapter.

If we get an unexpected response from the server, the sample app logs a message to Logcat and otherwise quietly fails. A production-grade app would handle these scenarios better, including informing the user about the problem.

Of course, a production-grade backup implementation might want more than what we have here, such as better security. For apps being publicly distributed through the Play Store or similar channels, you may want to offer multiple ways of saving off the backup, through some common API with multiple implementations. That way, users can choose whether to back up data via a private server or a public one (e.g., Amazon S3) or some other means that you offer.

Initiating a Restore

Unfortunately, on occasion, the user may have a need to restore the app's data from a backup.

There are three primary possible triggers for this work to be done:

- The user could ask for data to be restored manually, through some option in the app's UI, such as an action bar item
- The request to restore the data could be pushed to the device, such as through GCM, perhaps in response to an IT department staff member initiating a remote restore

- The user could have gotten a new device, and if the user had chosen automatic disaster recovery "backups" on their old device, they could have our app and its data automatically restored onto the new device

There is also the question of *which* backup to restore. Frequently, the user will want the most recent backup, but that is not always the case. The user might realize that the data has been wrong for days and needs to restore an earlier backup than the most recent one.

To that end, the `BackupClient` demo app will allow the user to manually request that data be restored, via a "Restore" action bar item. We will fetch a list of available backups from the backup server, so the user can choose what backup to restore from.

The "Restore" action bar item in `MainActivity` simply launches a `RestoreRosterActivity`, to allow the user to choose the backup to restore. That activity merely sets up a dynamic fragment, `RestoreRosterFragment`, in `onCreate()`:

```
package com.commonsware.android.backup;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class RestoreRosterActivity extends FragmentActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager()
      .findFragmentById(android.R.id.content)==null) {
      getSupportFragmentManager().beginTransaction()
        .add(android.R.id.content,
          new RestoreRosterFragment()).commit();
    }
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterActivity.java)

`RestoreRosterFragment` has fairly basic implementations of the `onCreate()`, `onStart()`, and `onStop()` lifecycle methods, to mark the fragment as being a retained fragment, plus to register and unregister from the event bus:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);
}

@Override
public void onStart() {
  super.onStart();

  EventBus.getDefault().register(this);
}

@Override
public void onStop() {
  EventBus.getDefault().unregister(this);

  super.onStop();
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

`RestoreRosterFragment` is a `ListFragment`, so the `ListView` will be set up automatically in the inherited implementation of `onCreateView()`. In `onViewCreated()`, we can kick off a REST request to pull down the list of backups from the backup server. This client assumes that the REST server has an `/api/backups` endpoint that will return a JSON roster of the available backups, so we can use OkHttp to perform the `GET` request for that data:

```
@Override
public void onViewCreated(View view,
                          Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  Request request=new Request.Builder()
    .url(URL_BACKUPS)
    .build();

  BackupService.OKHTTP_CLIENT.newCall(request).enqueue(this);
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

Here, we use the same `OkHttpClient` instance as `BackupService` uses — since this is a `static` data member that is automatically initialized, it does not matter whether

or not we have used `BackupService` already in this process. The endpoint URL is found in the `URL_BACKUPS` constant:

```
private static final String URL_BACKUPS=
  BuildConfig.URL_SERVER+"/api/backups";
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

Since this is being driven by the UI, and we are calling OkHttp from the main application thread, we use `enqueue()` instead of `execute()`, to schedule the request to be performed on a background thread supplied and managed by OkHttp. `RestoreRosterFragment` implements the required `Callback` interface needed by `enqueue()`. That interface, in turn, requires two methods. One is `onFailure()`, to be called if there is a problem in executing the HTTP request. Here, we just inform the user about the problem in a `Toast`, though a production-grade app would do something more sophisticated:

```
@Override
public void onFailure(Request request, IOException e) {
  Toast.makeText(getActivity(), R.string.msg_roster_failure,
    Toast.LENGTH_LONG).show();
  Log.e(getClass().getSimpleName(),
    "Exception retrieving backup roster", e);
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

The more important method is `onResponse()`, called when we get a valid-looking response from the server:

```
@Override
public void onResponse(Response response) throws IOException {
  Gson gson=new GsonBuilder()
    .setDateFormat("yyyy-MM-dd'T'HH:mm:ssZZZZZ")
    .create();

  Type listType=new TypeToken<List<BackupMetadata>>() {}.getType();

  EventBus
    .getDefault()
    .post(
      gson.fromJson(response.body().charStream(), listType));
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

This sample *could* use Retrofit for performing this REST-style `GET` request, in which case Retrofit would work with OkHttp and Google's Gson to parse our response. In this case, we are using OkHttp directly, and so we need to arrange to have Gson parse the response.

To that end, we:

- Create a `Gson` instance through a `GsonBuilder`, teaching it that the JSON data to be mapped to `Date` objects in our results have a particular serialized format
- Create a `Type` object wrapping our expected response: a `List` of `BackupMetadata` objects
- Get the JSON from the response (`response.body().charStream`), and pass that to the `Gson` object for parsing

And, since `onResponse()` is called on a background thread, we use the event bus to deliver that `List` of `BackupMetadata` objects to the fragment itself, so we can pick up that event on the main application thread.

The JSON we get back will be a JSON array containing a list of JSON objects, with each of those objects being mapped to a `BackupMetadata` instance by Gson:

```
package com.commonsware.android.backup;

import java.util.Date;

public class BackupMetadata {
  Date timestamp;
  String dataset;

  @Override
  public String toString() {
    return(timestamp.toString());
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/BackupMetadata.java)

`RestoreRosterFragment` then has an `onEventMainThread()` method, to pick up the `List` of `BackupMetadata`, to wrap that in an `ArrayAdapter` and put those results in the fragment's `ListView`:

```
  @Subscribe(threadMode =ThreadMode.MAIN)
  public void onEventMainThread(List<BackupMetadata> roster) {
```

```
    adapter=new ArrayAdapter<BackupMetadata>(getActivity(),
      android.R.layout.simple_list_item_1, roster);

    setListAdapter(adapter);
  }
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

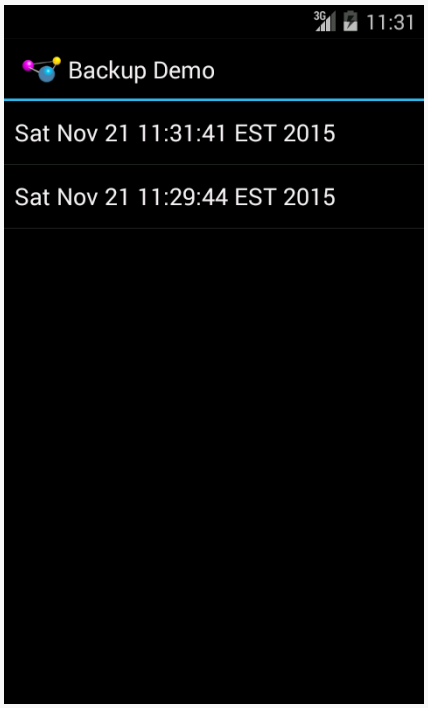


Figure 804: `RestoreRosterFragment`, Showing Two Backups

Starting the Restore Activity

When the user clicks on an available backup in the `ListView`, `onListItemClick()` gets called:

```
@Override
public void onListItemClick(ListView l, View v, int position,
                            long id) {
  String url=
    BuildConfig.URL_SERVER+adapter.getItem(position).dataset;
  Intent i=
    new Intent(getActivity(), RestoreProgressActivity.class)
      .setData(Uri.parse(url))
      .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|
```

```
                Intent.FLAG_ACTIVITY_CLEAR_TASK|
                Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);

    startActivity(i);
  }
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreRosterFragment.java)

The `BackupMetadata` has a relative URL to the backup's dataset, so we combine that with `BuildConfig.URL_SERVER` to get a fully-qualified URL. Then, we start up a `RestoreProgressActivity`, which will be responsible for kicking off the restore and showing some form of progress indicator along the way.

The tricky part with restoring your app's data is that you cannot have any app components running that rely upon that data, as the data will be changing out from underneath those components. In our case, we need to get rid of our `MainActivity`.

To do that, we attach a few flags to the `Intent` used to start up the `RestoreProgressActivity`:

- `FLAG_ACTIVITY_NEW_TASK`
- `FLAG_ACTIVITY_CLEAR_TASK`
- `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS`

These will get rid of all of our previous activities (including the currently-active `RestoreRosterActivity`) and will prevent the `RestoreProgressActivity` from showing up in the overview screen.

`RestoreProgressActivity` has a simple layout with a large centered `ProgressBar`:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ProgressBar
    style="@android:style/Widget.ProgressBar.Large"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"/>
</FrameLayout>
```

(from Backup/BackupClient/app/src/main/res/layout/progress.xml)

In `onCreate()` of `RestoreProgressActivity`, in addition to showing that `ProgressBar`, we kick off a `RestoreService` to actually download and restore the backup. We are passed the URL to the backup dataset in the `Intent` used to start `RestoreProgressActivity`, and we just pass that same URL along (as a `Uri`) to the service:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.progress);

  if (savedInstanceState==null) {
    RestoreService.enqueueWork(this);
  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreProgressActivity.java)

However, we only do that if we are not being recreated after a configuration change, so this only happens on the first invocation of the activity.

`RestoreProgressActivity` also registers for events on the event bus, using the typical `onStop()`/`onStart()` pattern:

```
@Override
protected void onStart() {
  super.onStart();

  EventBus.getDefault().register(this);
}

@Override
protected void onStop() {
  EventBus.getDefault().unregister(this);

  super.onStop();
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreProgressActivity.java)

Downloading and Restoring the Dataset

Meanwhile, over in `RestoreService`, we download and unpack the dataset:

```
package com.commonsware.android.backup;

import android.content.Context;
import android.content.Intent;
import android.support.annotation.NonNull;
import android.support.v4.app.JobIntentService;
import android.util.Log;
import com.squareup.okhttp.Request;
import com.squareup.okhttp.Response;
import org.greenrobot.eventbus.EventBus;
import java.io.File;
import okio.BufferedSink;
import okio.Okio;

public class RestoreService extends JobIntentService {
  private static final int UNIQUE_JOB_ID=1337;

  static void enqueueWork(Context ctxt) {
    enqueueWork(ctxt, RestoreService.class, UNIQUE_JOB_ID,
      new Intent(ctxt, RestoreService.class));
  }

  @Override
  public void onHandleWork(@NonNull Intent i) {
    Request request=new Request.Builder()
      .url(i.getData().toString())
      .build();

    try {
      Response response=
        BackupService.OKHTTP_CLIENT.newCall(request).execute();
      File toRestore=new File(getCacheDir(), "backup.zip");

      if (toRestore.exists()) {
        toRestore.delete();
      }

      BufferedSink sink = Okio.buffer(Okio.sink(toRestore));

      sink.writeAll(response.body().source());
      sink.close();

      ZipUtils.unzip(toRestore, getFilesDir(),
        BackupService.ZIP_PREFIX_FILES);
      ZipUtils.unzip(toRestore,
        BackupService.getSharedPrefsDir(this),
        BackupService.ZIP_PREFIX_PREFS);
      ZipUtils.unzip(toRestore, getExternalFilesDir(null),
```

```
        BackupService.ZIP_PREFIX_EXTERNAL);

      EventBus.getDefault().post(new RestoreCompletedEvent());
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
        "Exception restoring backup", e);
      EventBus.getDefault().post(new RestoreFailedEvent());
    }
  }

  static class RestoreCompletedEvent {

  }

  static class RestoreFailedEvent {

  }
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreService.java)

The URL for the dataset is coming in via the `Intent` passed into `onHandleIntent()`. We use that to build the OkHttp `Request`, then do a synchronous call via `execute()` to get the `Response`.

Previous uses of OkHttp in this chapter focused on REST responses, where we could either just use `Location` headers or pass the text of the response over to Gson. Here, we are expecting a ZIP file, and possibly a large one. The right way to get that written to disk (so we can unpack it) is to stream the data down and write that data out to disk, rather than attempting to read everything into memory first.

To that end, we take advantage of the fact that OkHttp itself is built atop Square's Okio library, which offers a nice Java API for handling streams, based on sinks and sources. The recipe for streaming an HTTP response to disk involves:

- Creating a sink for the destination file (in this case, a `backup.zip` file placed in `getCacheDir()`)
- Wrapping that in a `BufferedSink`
- Telling the sink to write everything from the `source()` we get from OkHttp representing the ZIP data
- Closing the sink

At that point, we need to unpack the dataset into the places we got the data from in the first place when we backed it up:

- `getFilesDir()`
- the directory for `SharedPreferences`
- `getExternalFilesDir()`

To that end, we use a slightly modified version of the `ZipUtils` class first referenced in the tutorials. The one used in the tutorials comes from the CWAC-Security library. However, that `ZipUtils` class does not handle two things that we need here:

- Unpacking a subset of the files, from one virtual directory within the ZIP archive
- Restoring them to an already-existing directory without deleting and recreating that directory.

The `BackupClient` project has its own modified version of `ZipUtils` that handles those cases. Beyond that, the `unzip()` method is the same as before, taking:

- The ZIP file to unpack
- The filesystem directory where the unpacked files should go
- The virtual directory within the ZIP archive that we want (as opposed to the entire contents)

When that is done, we post a `RestoreCompletedEvent`. If there is some problem, we post a `RestoreFailedEvent`, in addition to logging details to Logcat.

`RestoreProgressActivity` listens for both of those events:

```
@Subscribe(threadMode =ThreadMode.MAIN)
public void onCompleted(RestoreService.RestoreCompletedEvent event) {
  startActivity(new Intent(this, MainActivity.class));
  finish();
}

@Subscribe(threadMode =ThreadMode.MAIN)
public void onFailed(RestoreService.RestoreFailedEvent event) {
  Toast.makeText(this, R.string.msg_restore_failed,
    Toast.LENGTH_LONG).show();
  finish();
}
```

(from Backup/BackupClient/app/src/main/java/com/commonsware/android/backup/RestoreProgressActivity.java)

In the success case, we can now start up a fresh `MainActivity` (since the original was destroyed as part of launching `RestoreProgressActivity`), and it can read the restored data.

In the failure case... we are really screwed. We may have partially restored the data, but perhaps not all of it, and there is no telling what state the data is in. A production-grade app would handle this by:

- Moving all of the existing data to a safe location on the device
- Attempting to restore the data
- If there is an unhandled exception in the restoration process, deleting the partially-restored data and moving the original data back into position

This would reduce the odds of some catastrophic problem wiping out the app. In this sample, though, we just show a `Toast`, `finish()` the activity (thereby exiting the app, as we have no other active activities), and hoping the user uninstalls and reinstalls the app, or just uninstalls the app, or something.

Trying This Yourself... With a Little Help from Ol' Blue Eyes

Everything discussed so far assumes the existence of some REST-style Web server that we can interact with for backups. As it so happens, the `BackupClient` project has a crude implementation of such a server, in the form of a Ruby script using the Sinatra gem:

```
require 'fileutils'
require 'time'
require 'sinatra'
require 'json'

BACKUP_ROOT='/tmp/backups'

get '/' do
  'Hello world!'
end

get '/api/backups' do
  result=[]

  if File.exist?(BACKUP_ROOT)
    Dir.foreach(BACKUP_ROOT) do |item|
      next if item == '.' or item == '..'

      subdir=File.join(BACKUP_ROOT, item)
```

```
      if File.directory?(subdir)
        f=File.join(subdir, "metadata.json")

        if File.exist?(f)
          metadata=JSON.load(open(f))
          metadata['dataset']="/api/backups/#{item}/dataset"

          result << metadata
        end
      end
    end
  end

  result.sort_by!{|metadata| metadata['timestamp']}
  result.reverse!

  JSON.pretty_generate(result)
end

post '/api/backups' do
  id=SecureRandom.uuid
  dir=File.join(BACKUP_ROOT, id)
  FileUtils.mkdir_p(dir)
  f=File.join(dir, "metadata.json")
  metadata={'timestamp'=>Time.new.xmlschema}
  File.open(f, 'w') {|io| io.write(JSON.generate(metadata))}

  redirect to('/api/backups/'+id), 201
end

put '/api/backups/:id/dataset' do
  dir=File.join(BACKUP_ROOT, params[:id])

  if File.exist?(dir)
    f=File.join(dir, "backup.zip")
    File.open(f, 'w') {|io| io.write(request.body.read)}

    redirect to("/api/backups/#{params[:id]}/dataset"), 201
  else
    status 404
  end
end

get '/api/backups/:id/dataset' do
  dir=File.join(BACKUP_ROOT, params[:id])
  f=File.join(dir, "backup.zip")
```

```
  if File.exist?(f)
    send_file f
  else
    status 404
  end
end
```

(from Backup/BackupClient/server.rb)

If you have familiarity with Ruby, you can:

- install the `sinatra` and `json` gems in your environment
- run the script (`ruby server.rb`)

That will give you a server, listening to `localhost:4567`... which happens to be what the `BackupClient` Android app is looking to talk to, if that app is running on an emulator. If you want to test with an actual Android device, the `-o` switch lets you specify the IP address to listen to, and `-p` lets you change up the port number if you wish.

The Google Backup Bootstrap

Once you get your real backup system going, then, if you wish, you can play around with Google's disaster recovery bootstrap. By opting into what Google terms "backup", you can have some of your data automatically backed up, then restored when the user replaces their device.

What to Bootstrap?

The biggest decision that you will need to make is what should be included in Google's bootstrap backup and what should not.

The primary considerations are privacy and security. Any data included in the bootstrap is visible to other parties. If that data is not encrypted with a user-supplied passphrase, other parties will be able to do what they want with the data, without much recourse.

One option, therefore, is to opt out of these bootstrap backups entirely, and handle disaster recovery like any other restore process.

Another is to only include some identifying information in the bootstrap backup, to help expedite the restore process, but without really compromising security much.

In the context of the `BackupClient` sample shown earlier in this chapter, if the backup server was adequately secured, including a dataset URL in the bootstrap backup would not be much of a problem. Having the URL itself is probably not that useful, and if only authorized users can download datasets from those URLs, attackers would not gain anything from peeking at the bootstrap. `BackupClient` itself has very little security, to keep the sample (reasonably) simple, but you can imagine requiring user accounts or similar means to try to lock down access to the backup server.

The far other end of the spectrum is to allow Android to backup "the whole shootin' match" (i.e., everything), on the grounds that the data you have is not especially private.

You and your qualified legal counsel will need to make this decision before deciding what to do for implementing the bootstrap backup itself.

Bootstrap Backup on Android 6.0+

Android has had a backup API since Android 2.2. However, not only did developers have to opt into the backups, but they had to write special code to assist in those backups. As such, that API was not used that much.

Android 6.0 has gone the other direction, with opt-out backups of all likely data, if your `targetSdkVersion` is 23 or higher. Specifically:

- Your app's internal storage (`getFilesDir()`, `SharedPreferences`, `getDatabaseDir()`, etc.) gets backed up, with the exception of `getCacheDir()` and `getNoBackupFilesDir()` (the latter introduced in API Level 21)
- `getExternalFilesDir()` is backed up, but not other locations on external storage

Backups occur approximately once per day, if the device is idle, charging, and on WiFi.

Configuring the Backup

If what you want to back up is different than what Android 6.0+ will back up by default, you can add manifest entries to better control what is and is not backed up.

To opt out entirely, add `android:allowBackup="false"` to your `<application>` element in the manifest:

```
<application
  android:allowBackup="false"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme"
  tools:replace="android:allowBackup">
  <!-- other cool stuff here -->
</application>
```

Here, the `tools:replace` ensures that no library attempts to override your `allowBackup` value.

Conversely, if you want to participate in the bootstrap backup, but you want to change the roster of what gets backed up, use the `android:fullBackupContent` attribute on the `<application>` element. This needs to point to an XML resource that describes what it is that you do and do not want backed up.

The `BackupClient` sample has this configured. The `<application>` element points to a `res/xml/backup_rules.xml` resource:

```
  <application
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity android:name=".RestoreRosterActivity" />
    <activity android:name=".RestoreProgressActivity" />

    <service
      android:name=".BackupService"
      android:permission="android.permission.BIND_JOB_SERVICE" />
    <service
      android:name=".RestoreService"
```

```
      android:permission="android.permission.BIND_JOB_SERVICE" />
  </application>
```

(from Backup/BackupClient/app/src/main/AndroidManifest.xml)

That XML resource can contain `<include>` and `<exclude>` elements, inside of a root `<full-backup-content>` element. The rules are:

- If there are no `<include>` elements — only `<exclude>` elements – then all the files that get backed up by default will get backed up, except those blocked by those `<exclude>` elements.
- If there are one or more `<include>` elements (perhaps along with `<exclude>` elements), then *none* of the files that get backed up by default will be backed up. Instead, only the files listed in the `<include>` elements (and not blocked by any `<exclude>` elements) will be backed up.

The `BackupClient` sample has two `<include>` elements, in effect saying that only what is cited in these elements should be backed up:

```
<?xml version="1.0" encoding="utf-8"?>
<full-backup-content>
  <include
    domain="sharedpref"
    path="com.commonsware.android.backup_preferences.xml"/>
  <include
    domain="sharedpref"
    path="com.commonsware.android.backup.BackupService.xml"/>
</full-backup-content>
```

(from Backup/BackupClient/app/src/main/res/xml/backup_rules.xml)

The `<include>` and `<exclude>` elements must have a `domain` attribute and a `path` attribute. These combine to indicate what is being included or excluded.

The `domain` attribute indicates one of five locations relative to your app:

- `root` points to all of your internal storage
- `file` points to the subset of your internal storage used for ordinary files (i.e., `getFilesDir()`)
- `database` points to the subset of your internal storage used for databases (i.e., `getDatabasePath()`)
- `sharedpref` points to the subset of your internal storage used for `SharedPreferences`
- `external` points to the location used by `getExternalFilesDir(null)`

The path attribute then provides a relative path, from the base location indicated by domain, for the item to be included or excluded.

Hence, the BackupClient backup rules say to include two SharedPreferences files. One is written to by BackupService on every backup, holding a single value, keyed by lastBackupDataset, with the URL to the last backup dataset. The other is the default SharedPreferences, used for the last-visited tab by the UI. Because these SharedPreferences files are included in the bootstrap backup, they should be restored in case the user replaces the device. However, they are the *only* things that is supposed to be backed up — everything else in the app should be left alone.

Note that the documentation does not state clearly if the path attribute is required. It is possible that the path attribute is optional, where if it is missing, it means you want to include or exclude everything in the cited domain.

Testing the Backup and Restore Steps

In theory, to test your backup configuration, you can run three commands on the command line:

```
adb shell setprop log.tag.BackupXmlParserLogging VERBOSE
adb shell bmgr run
adb shell bmgr fullbackup ...
```

where ... is the application ID of the app to be backed up. For the sample app, that is com.commonsware.android.backup.

(the above assumes that you have adb in your PATH)

You can then manually initiate a restore operation via:

```
adb shell bmgr restore ...
```

for the same value of .... Presumably, you would do this after modifying or clearing the backed-up data, so you can confirm that the data was restored properly.

For the purposes of conducting lightweight experiments with the auto-backup facility, you do not need to mess around with the entire backup system outlined earlier in this chapter. That backs up the actual content; the auto-backup facility is backing up SharedPreferences, and that happens whether or not we are also backing up the content.

So, for example, you could do the following:

- Run the sample app and switch to some tab other than the default, then press BACK to exit the app and save your last-visited tab in `SharedPreferences`
- Execute the following at the command line of your developer machine, to examine the contents of the `SharedPreferences`:

```
adb shell run-as com.commonsware.android.backup
"cat /data/data/com.commonsware.android.backup/
shared_prefs/com.commonsware.android.backup_preferences.xml"
```

(**NOTE**: the above should be all on one line; it is split here across three lines due to the length of the command)

You should see something like:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <int name="lastVisited" value="2" />
</map>
```

The `value` will be the index of whatever tab you were on when you exited the activity.

- Run the commands shown earlier to back up those `SharedPreferences`:

```
adb shell setprop log.tag.BackupXmlParserLogging VERBOSE
adb shell bmgr run
adb shell bmgr fullbackup com.commonsware.android.backup
```

You should see output in Logcat indicating that the backup was taken:

```
14936-14936/? D/AndroidRuntime: Calling main entry com.android.commands.bmgr.Bmgr
800-2345/? D/BackupManagerService: fullTransportBackup()
800-14960/? I/PFTBT: Initiating full-data transport backup of ...
800-14961/? D/BackupManagerService: Binding to full backup agent : ...
800-14961/? D/BackupManagerService: awaiting agent for ApplicationInfo{...}
800-810/? D/BackupManagerService: agentConnected pkg=com.commonsware...
800-14961/? I/BackupManagerService: got agent
android.app.IBackupAgent$Stub$Proxy@e17804c
800-14961/? I/BackupRestoreController: Getting widget state for user: 0
800-14962/? I/file_backup_helper:    Name: apps/com.commonsware.android...
800-14962/? D/BackupManagerService: Calling doFullBackup() on com.commonsware...
```

```
9380-9391/com.commonsware.android.backup I/file_backup_helper:    Name: ...
800-14960/? I/PFTBT: Transport suggested backoff=0
800-14960/? I/PFTBT: Full backup completed.
9380-9380/? I/Process: Sending signal. PID: 9380 SIG: 9
800-2345/? D/BackupManagerService: Done with full transport backup.
```

(**NOTE**: the lines have been truncated due to length)

- Run the app again and switch to another tab, then press BACK to exit the activity.
- Run the `run-as` command again to examine the contents of the current `SharedPreferences`, and see that it contains your newly-chosen tab.
- Execute `adb shell bmgr restore com.commonsware.android.backup` from the command line to restore the `SharedPreferences` from your backup. You should get additional lines in Logcat showing that the restoration took place:

```
16814-16814/? D/AndroidRuntime: Calling main entry com.android.commands.bmgr.Bmgr
800-7101/? V/BackupManagerService: beginRestoreSession: pkg=com.commonsware...
800-2345/? V/RestoreSession: restorePackage pkg=com.commonsware.android.backup ...
800-2345/? V/RestoreSession: restorePackage pkg=com.commonsware.android.backup ...
800-1111/? D/BackupManagerService: MSG_RUN_RESTORE observer=android.app.backup...
800-1111/? D/BackupManagerService: initiateOneRestore packageName=@pm@
800-1111/? E/SELinux: SELinux: Could not get canonical path /cache/@pm@.restore ...
800-1111/? I/BackupManagerService: Next restore package: RestoreDescription{...}
800-16839/? I/RestoreEngine: Sig + version match; taking data
800-16839/? D/RestoreEngine: Need to launch agent for com.commonsware.android.backup
800-16839/? D/RestoreEngine: Clearing app data preparatory to full restore
800-16839/? I/ActivityManager: Force stopping com.commonsware.android.backup ...
800-16839/? I/ActivityManager: Killing 15029:com.commonsware.android.backup/...
800-1195/? D/GraphicsStats: Buffer count: 5
800-1198/? W/ActivityManager: Spurious death for ProcessRecord...
5005-5986/? D/Documents: Update found 7 roots in 8ms
1888-16840/? D/PackageBroadcastService: Received broadcast ...
1888-16840/? D/AccountUtils: Clearing selected account for com.commonsware...
1888-16840/? I/LocationSettingsChecker: Removing dialog suppression flag...
1888-2082/? I/Icing: doRemovePackageData com.commonsware.android.backup
800-16839/? I/ActivityManager: Start proc 16848:com.commonsware.android...
800-16839/? D/BackupManagerService: awaiting agent for ApplicationInfo{...}
16848-16848/? I/art: Late-enabling -Xcheck:jni
16848-16848/? W/System: ClassLoader referenced unknown path: ...
800-1128/? D/BackupManagerService: agentConnected pkg=com.commonsware.android...
800-16839/? I/BackupManagerService: got agent android.app...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
```

```
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: Final tally.
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: Includes:
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging:   domain=sp
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: Excludes:
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging:   ...nothing to
exclude.
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging:
16848-16865/com.commonsware.android.backup V/BackupXmlParserLogging: ...
800-1111/? V/BackupManagerService: No more packages; finishing restore
800-2345/? D/RestoreSession: endRestoreSession
800-1111/? I/BackupRestoreController: restoreFinished for 0
800-1111/? I/BackupManagerService: Restore complete.
800-1111/? V/BackupManagerService: Clearing restore session and halting timeout
```

- Run the `run-as` command again to examine the contents of the current `SharedPreferences`, and see that it contains your original tab.
- Uninstall your application, then try the `run-as` again, which will give you an error indicating that the file was not found.
- Re-run the app from the IDE. Then, run the `run-as` command again, to see that your file was restored without manually having to restore it.

Bootstrap Backup on Android 2.2-5.1

Prior to Android 6.0, Android had a "backup service", inaugurated in Android 2.2. As with the Android 6.0 approach, the original backup service was mostly for disaster recovery.

Unlike with Android 6.0's approach, you needed to opt into having these backups. Partly, this opt-in was accomplished via code, as you had to extend a `BackupAgentHelper` and register it in your manifest as the `android:backupAgent`, via the `<application>` element. The `BackupAgentHelper` subclass would indicate what should be backed up, by instantiating one or more `BackupHelper` objects (e.g., `FileBackupHelper`), configuring them to back up certain items, then registering them with the `BackupAgentHelper` via an `addHelper()` method.

Partly, though, the opt-in was accomplished via registering for an API key. This process was never integrated into the rest of the Play Services architecture, which now has a standardized approach for registering for various API keys and agreeing to the terms of service for each.

Instead, you would need to visit an obscure Web page, agree to the terms of service, provide information about your app (notably the application ID), get the API key, and add it to your manifest via a `<meta-data>` element.

However, those terms of service contain some interesting clauses, ones that may give your legal counsel some concern, such as:

- "the form and nature" of the backup service "may change from time to time without prior notice to you"
- "Google may stop... providing the Service (or any features within the Service) to you or to users generally at Google's sole discretion, without prior notice to you"
- You "agree to use the Service only for purposes that are permitted by... any applicable law... in the relevant jurisdictions (including any laws regarding the export of data or software to and from the United States or other relevant countries)" without ever disclosing what those "relevant countries" are
- You agree to not "sell... access to the Service", which would seem to preclude its use by paid apps or apps using in-app purchases to upgrade to some "pro" edition that enabled backups
- "you are responsible for maintaining the security... of the Backup Service Key(s)", despite the fact that these have to be published in the manifest and therefore are readable by anyone
- "you will not transmit any Content through the Service that is copyrighted, protected by trade secret or otherwise subject to third party proprietary rights", despite the fact that developers have no means of validating these rights for user-supplied content
- "Google may need to change these Terms from time to time... Once the modified Terms are posted, the changes will become effective immediately, and you are deemed to have accepted the modified Terms if you continue to use the Service", despite the fact that developers have no means of finding out exactly when the terms change or somehow instantaneously preventing installed copies of their apps from using the service

Beyond this, there are no statements about where the data is actually backed up, other than opening it up to just about anyone that Google wishes to characterize as "Subsidiaries and Affiliates".

Please discuss these terms with legal counsel before registering for this service and integrating it within your app.

Additional documentation about this form of backup, should you choose to pursue it, can be found online.

Boosting Backup Security

Backups, in effect, are intentional data leaks. You want something other than the device to have access to your app's data. Hence, it is important to take reasonable steps to ensure that those backups are secure, secure enough that nobody is going to be able to exploit them for uses that go against the user's wishes. Rest assured that people will try to exploit backups and will succeed if your security is insufficient.

Securing Access to the Dataset

The backup dataset that you transfer off the device needs to be secure from attack. Unauthorized people should not be able to get at the dataset.

For a backup system like the one outlined in this chapter, the big thing to secure is access to the dataset via its URL. If anyone who gets the URL can download the dataset, now all an attacker needs to do is determine how to get that URL, such as by exploiting flaws in Google's bootstrap backup. Or, for that matter, Google staff could get at the URL, at least in principle.

In this case, the URL alone must be insufficient. It would need to be combined with other information from the user, such as some sort of site authentication, where that other information is not retained.

If you are holding onto backup datasets yourself, on your own servers, you will also need to ensure that only authorized staff can get at those datasets and that such access is highly visible. Otherwise, you are at risk of an insider attack, whether through so-called "social engineering" or just good old-fashioned extortion.

Securing Transmission of the Dataset

Another way that an attacker could get at the dataset is to copy the data in motion, as it is sent from your app to the backup server. Make sure that you are using suitable security here:

- HTTPS with certificate pinning
- Corporate VPN
- etc.

Bear in mind that users may wind up making a backup from any sort of network, ranging from your office network to the free WiFi at a local coffee shop. In principle, you could detect this and refuse to back up the data when you do not recognize the network. However, this reduces the value of the backup system, as the user might not be able to make a manual backup at some point when they need it (e.g., on business travel).

Encrypting the Dataset

The ultimate in protection for the user is to have the data be encrypted by a user-supplied passphrase. Then, even *you* cannot access the data without the user's assistance. There are ways of addressing this, perhaps involving brute force attacks or other sorts of brute force attacks. However, it certainly slows attackers down.

The simplest way to have encrypted backups — from the standpoint of the person writing the backup code — is to encrypt the data itself. For example, you do not necessarily need to re-encrypt a SQLCipher for Android database as part of a backup dataset, as it is already encrypted. Note, though, that having encrypted data at rest does not mean you can skip encrypting the data in motion, as it is sent to your backup server. While attackers would not be able to *read* the backed-up data readily, they could *replace* the backed-up data sent over the unencrypted communications channel and perhaps cause problems that way.

If, however, you are not in position to encrypt the data at rest within your app, you may wish to consider asking the user for a passphrase and using that to encrypt the backup dataset. Note that this passphrase requirement largely eliminates the ability for you to do unattended automated backups, as you either do not have the passphrase then (and so cannot encrypt the backups) or you are *saving* the passphrase (and so have just made it trivial for somebody to get it and decrypt the data).

Alternative Approaches

Backing up local data is essential where the device is the system of record, to be able to deal with catastrophe (e.g., the user accidentally uninstalls the app).

That being said, there are a few ways of dealing with backing up local data that might not necessarily seem to the user as though it is a backup process.

Data Versioning

Beyond the accidental wiping of data, such as through an erroneous install, a backup can also help recover from more fine-grained errors, like accidentally deleting a bit of data (e.g., a row or set of rows from database tables).

One way to address that is to use some sort of data versioning approach. Many software developers are familiar with this in the form of source code version control, such as `git`. Here, you never really "delete" anything forever. Instead, you delete (or change) things in your working copy of the data, with the versioning system tracking changes to the data, so you can roll back to some earlier version if the need arises.

This is not limited to source code or similar sorts of documents. One simple example of versioning that has been used for decades is to not actually delete database rows, but instead set some `is_deleted` column to a known value. Then, when you query the database, you filter out the "deleted" rows by excluding from the query those rows where `is_deleted` is set to that specific value. Recovering those deleted rows is then a matter of showing all the deleted ones to the user and clearing `is_deleted` for the ones to be restored.

Obviously, this gets much more complicated once you get into foreign key constraints (i.e., how can you restore X if it depends on Y that was also deleted?). And it is not a full replacement for a backup-and-restore system, since anything that damages or deletes the entire database cannot be recovered via this sort of versioning. But, if you are looking to implement a robust disk-based "undo" facility for users, just bear in mind that it also helps out for some sorts of cases where you might ordinarily think of restoring from a backup.

Import and Export

Another feature that you can add that has some relationship to data backups is data import and export. Whatever is exported can be backed up by the user by some other means; if the master copy of the app's data gets damaged, you might be able to recover from that damage via importing a previous export.

Of course, import and export are also used for data exchange with foreign systems (e.g., exporting tabular data in a format that can be read in by a desktop spreadsheet program). Also, traditionally, import and export are tasks that are manually requested by users. However, you might consider giving the user an option of performing an automatic export as a replacement for, or adjunct to, some other form of regular backup.

Data Synchronization

The ultimate solution for not having to mess with a robust device-based backup system is to not have the device be the system of record. Instead, some server is the system of record, with the device holding what amounts to a persistent cache of some of that data:

- Data that you retrieved previously, so you do not necessarily have to keep downloading the same data from the server
- Data that the user has modified that you are planning on sending to the server at some time in the future (e.g., during the nightly sync, or when the Internet is available again).

Trail: Security

SSL

The traditional approach to securing HTTP operations is by means of SSL. Android supports SSL, much as ordinary Java does. Most of the time, you can just allow Android to do its thing with respect to SSL, and you will be fine. However, there may be times when you have to play a more direct role in SSL communications, to handle arbitrary SSL-encrypted endpoints, or to help ensure that your app is not the victim of a man-in-the-middle attack.

This chapter will explore various SSL scenarios and how to address them.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on Internet access.

Basic SSL Operation

Generally speaking, SSL "just works", for ordinary sites with ordinary certificates.

If you use an `https:` URL with `HttpUrlConnection` or `WebView`, SSL handshaking will happen automatically, and assuming the certificates check out OK, you will get your result, just as if you had requested an `http:` URL.

However, originally, requesting a download via `DownloadManager` with an `https:` scheme would result in `java.lang.IllegalArgumentException: Can only download HTTP URIs`. As of Android 4.0, SSL is supported. Hence, you need to be careful about making SSL requests via `DownloadManager` if your `minSdkVersion` is less than 14.

For example, the Retrofit and Picasso sample apps from the chapter on Internet access both use `https://api.stackexchange.com` for their service endpoint. As a result, those requests — for the API JSON, at least — will go over SSL. You would need to log the URLs used for the image avatars to see whether StackExchange gives you `https` URLs or not.

Problems in Paradise

Ideally, SSL just works.

In practice, it often does, but depending on your app and your situation, you may encounter issues, such as:

- You want to test using SSL, but your test server does not have a domain name, let alone a SSL certificate, and so you need to try using a self-signed certificate
- Your IT department chose an obscure certificate authority for obtaining the SSL certificate used by your production server, and older Android devices do not recognize that certificate authority
- You are worried about MITM ("man-in-the-middle" or "Martian-in-the-middle") attacks, and you hear all these scary things about certificate authorities being hacked, and so you want to try to ensure that only valid certificates are honored by your app

And so on.

Here are some more details about some common SSL problems.

Self-Signed Certificate

SSL certificates used for public Web sites are usually backed by a "root certificate authority" that is well-known. That is not always the case.

One case is when the certificate is "self-signed", meaning that it was generated by somebody without involving a certificate authority. If you have shipped a production Android app, you created a self-signed certificate when you created your production key store. And you have been using a system-generated self-signed certificate throughout your development, known as the "debug signing key".

Self-signed certificates are rarely used on public-facing Web sites, as Web browsers are taught to warn users when such certificates are encountered. However, self-signed certificates might be used on internal servers, particularly test servers and other non-production environments.

There are even some benefits for using a self-signed certificate for production servers, if those servers will be talking only to your own apps and not arbitrary Web browsers.

Wildcard Certificate

Some certificates are difficult to validate because they use wildcards.

For example, Amazon S3 is a file storage and serving "cloud" solution from Amazon.com. They allow you to define "buckets" containing "objects", where each object then has its own URL. That URL is based on the name of the bucket and the name of the object. One option is for you to have the domain name of the URL be based on the name of the bucket, leaving the path to be solely the name of the object. This works, even with SSL, but Amazon needed to use a "wildcard SSL certificate", one that matches `*.s3.amazonaws.com`, not just a single domain name. By default, this will fail on Android, as Android's stock `TrustManager` will not validate wildcards for multiple domain name segments (e.g., `http://misc.commonsware.com.s3.amazonaws.com/foo.txt`). You will get an exception akin to:

```
javax.net.ssl.SSLHandshakeException: java.security.cert.CertificateException:
No subject alternative DNS name matching misc.commonsware.com.s3.amazonaws.com found
```

Custom Certificate Authority

Some larger organizations have set up their own certificate authority. Sometimes, they aspire to become a recognized root certificate authority, but have not been adopted by many browsers. Sometimes, they simply want to have more structure than a pure self-signed certificate but do not necessarily want to have all certificates go through a root certificate authority, perhaps due to expense.

In these cases, Android will reject the SSL certificate, for the same reason it rejects self-signed ones: it cannot validate the certificate chain all the way back to a known root certificate authority. But, with a little work, you can enable Android to support these as well.

Man in the Middle Attacks

Man-in-the-middle (MITM) attacks are a common way of trying to intercept SSL encrypted communications. The "man" in the "middle" might be a proxy server, a different Web site you wind up communicating with via DNS poisoning, etc. The objective of the "man" is to pretend to be the actual Web site or Web service you are trying to communicate with. If your app "falls for it", your app will open an encrypted channel to the attacker, not your site, and the attacker will have access to the unencrypted data you send over that channel.

Unfortunately, Android apps have a long history of being victims of man-in-the-middle attacks.

"Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security", an analysis of possible man-in-the-middle attacks on Android, is depressing. One in six surveyed apps explicitly ignored SSL certificate validation issues, mostly by means of do-nothing `TrustManager` implementations as noted above. Out of a selected 100 apps, 41 could be successfully attacked using man-in-the-middle techniques, yielding a treasure trove of credit card information, account credentials for all the major social networks, and so forth.

Their paper outlines a few ways in which apps can screw up SSL management — the following sections outline some of them.

Disabling SSL Certificate Validation

As mentioned above, if you disable SSL certificate validation, by implementing and using a do-nothing `TrustManager`, you are wide open for man-in-the-middle attacks. A simple transparent proxy server can pretend to be the real endpoint — apps ignoring SSL validation entirely will trust that the transparent proxy is the real endpoint and, therefore, perform SSL key exchange with the proxy rather than the real site. The proxy, as a result, gets access to everything the app sends.

Ignoring Domain Names

A related flaw is when you disable hostname verification. The "common name" (CN) of the SSL certificate should reflect the domain name being requested. Requesting `https://www.foo.com/something` and receiving an SSL certificate for `xkcdhatguy.com` would be indicative of a mis-configured Web server at best and a man-in-the-middle attack at worst.

By default, this is checked, and if there is no match, you will get errors like:

```
javax.net.ssl.SSLException: hostname in certificate didn't match: <...>
```

where the `...` is replaced by whatever domain name you were requesting.

But some developers disable this check. Perhaps during development they were accessing the server using a private IP address, and they were getting `SSLExceptions` when trying to access that server. It is very important to allow Android to check the hostname for you, which is the default behavior.

Hacked CAs

The truly scary issue is when the problem stems from the CA itself.

Comodo, TURKTRUST, and other certificate authorities have been hacked, where nefarious parties gained the ability to create arbitrary certificates backed by the CA. For example, in the TURKTRUST case, Google found that somebody had created a `*.google.com` certificate that had TURKTRUST as the root CA. Any browser — or Android app — that implicitly trusted TURKTRUST-issued certificates would believe that this certificate was genuine. This is the ultimate in man-in-the-middle attacks, as code that is ordinarily fairly well-written will believe the CA and therefore happily communicate with the attacker.

Even well-intentioned certificate authorities sometimes make mistakes. StartSSL offered a tool called StartEncrypt to make it easy to request and install certificates on a Web server. However, they made mistakes in the Web service API used by that tool to communicate back to StartSSL's servers. Attackers could create SSL certificates for a wide range of existing domains, including `google.com`, `facebook.com`, and other widely-used domains. Those fraudulent certificates could have been used to implement MITM attacks.

Introducing Network Security Configuration

You can use a "network security configuration" to help address those issues. This comes in the form of an XML resource, which you teach Android to use for your network connections. That resource tailors what you do and do not want to accept for SSL connections, such as "yes, I want to accept this self-signed certificate, at least for debug builds of the app" and "yes, I am willing to accept this additional certificate authority".

This XML resource will have a `<network-security-config>` root element. That in turn will contain:

- Zero or one `<base-config>` elements, defining global rules
- Zero, one, or several `<domain-config>` elements, defining rules to apply to a specific domain name or set of domain names
- Zero or one `<debug-overrides>` elements, defining global rules that will be applied only for debug builds of your app

The Native Android 7.0 Version

On Android 7.0 and higher, you can direct Android to apply your network security configuration by having an `android:networkSecurityConfig` attribute on the `<application>` element in your manifest:

```
<application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:networkSecurityConfig="@xml/net_security_config">
    // other stuff here
</application>
```

The name of the XML resource does not matter, so long as your `android:networkSecurityConfig` attribute points to it.

On Android 7.0 and higher, your network security configuration will be applied automatically for all network connections, without any Java configuration.

The CWAC-NetSecurity Backport

At the time of this writing, Google has not released an official backport of the network security configuration subsystem.

The author of this book converted that subsystem into a library – CWAC-NetSecurity — that serves as a backport, working back to API Level 17 (Android 4.2). It does not support every feature of the native implementation, and it requires a bit of Java code to arrange to use your network security configuration for HTTP requests. However, you can use the same XML resource structure. As with many backports, the vision is that you would use the backport until such time as your `minSdkVersion` rises to 24 or higher, at which point you can just use the native implementation.

The CWAC-NetSecurity library also offers a `TrustManagerBuilder` and related classes to make it easier for developers to integrate the network security configuration backport, particularly for OkHttp3 and `HttpURLConnection`.

The artifact for this library is distributed via the CWAC repository, so you will need to configure that in your module's `build.gradle` file, along with your `implementation` statement:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'com.commonsware.cwac:netsecurity:0.0.1'
    implementation 'com.squareup.okhttp3:okhttp:3.4.0'
}
```

If you are using this library with OkHttp3, you also need to have a `implementation` statement for a compatible OkHttp3 artifact, as shown above.

If you are using `HttpURLConnection`, or tying this code into some other HTTP client stack, you can skip the OkHttp3 dependency.

Next, add in this `<meta-data>` element to your manifest, as a child of the `<application>` element:

```
<meta-data
  android:name="android.security.net.config"
  android:resource="@xml/net_security_config" />
```

The value for `android:resource` should be the same XML resource that you used in the `android:networkSecurityConfig` attribute in the `<application>` element for the native network security configuration support on Android 7.0.

Then, in your code where you want to set up your network communications, create a `TrustManagerBuilder` and teach it to load the configuration from the manifest:

```
TrustManagerBuilder tmb=
  new TrustManagerBuilder().withManifestConfig(ctxt);
```

(where `ctxt` is some `Context`)

If you are using OkHttp3, create your basic `OkHttpClient.Builder`, then call:

```
OkHttp3Integrator.applyTo(tmb, okb);
```

(where `tmb` is the `TrustManagerBuilder` from before, and `okb` is your `OkHttpClient.Builder`)

At this point, you can create your `OkHttpClient` from the `Builder` and start using it.

If you are using `HttpURLConnection`, you can call `applyTo()` on the `TrustManagerBuilder` itself, passing in the `HttpURLConnection`. Afterwards, you can start using the `HttpURLConnection` to make your HTTP request:

```
      URL url=new URL(i.getData().toString());
      HttpURLConnection c=
        (HttpURLConnection)url.openConnection();
      TrustManagerBuilder tmb=
        new TrustManagerBuilder().withManifestConfig(this);

      tmb.applyTo(c);
```

(from Internet/CA/app/src/main/java/com/commonsware/android/downloader/Downloader.java)

In either case, on Android 7.0+ devices, `withManifestConfig()` will *not* use the backport. Instead, the platform-native implementation of the network security configuration subsystem will be used. On Android 4.2-6.0 devices, the backport will be used.

SSL Problems and Network Security Configuration

With all that as prologue, let's examine how the network security configuration subsystem — native or backport — can address some of the SSL issues outlined earlier in this chapter.

The sample code for these scenarios comes from the `Internet/CA` sample application. This application is based on some of the samples from the chapter on notifications, that use `HttpURLConnection` to download a PDF from the CommonsWare site.

Pinning the Certificate Authority

Your app may only communicate with one server, such as an employee-only server for your organization. To help limit the risk of possible MITM attacks, you might want to lock down your app, to only work with certificates coming from your chosen certificate authority for this server. That way, in addition to the other logistical problems facing attackers, they would need to get a forged SSL certificate from *your* certificate provider, instead of a forged SSL certificate from *any* certificate provider.

To make this work, first, you will need a PEM or DER file representing the root certificate for the certificate authority. Usually, the certificate authority will publish one of these on its Web site. You will need to put that file in `res/raw/` of your project, under a suitable resource name. For this scenario, in the sample app, there are two raw resources of note: `addtrustexternalcaroot.pem` and `verisign_class3.pem`, for Comodo and Verisign, respectively.

Next, you will need to create your network security configuration. As noted above, this is an XML resource, in `res/xml/`, that describes what changes you wish to make to the mix of supported certificate authorities. In the sample app, one such resource is `res/xml/network_comodo.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="false" usesCleartextTraffic="false">wares.commonsware.com</domain>
    <trust-anchors>
      <certificates src="@raw/addtrustexternalcaroot" />
    </trust-anchors>
  </domain-config>
</network-security-config>
```

(from Internet/CA/app/src/main/res/xml/network_comodo.xml)

As mentioned previously, the root element is `<network-security-config>`. In there, you can have one or more `<domain-config>` elements, describing the rules that you wish to apply to certain domains being used by your app.

A `<domain-config>` element will have one or more `<domain>` elements, listing domains that this particular configuration controls. Here, we have just one, for `wares.commonsware.com`. The `includeSubdomains` attribute indicates whether this rule applies to subdomains of the base domain, such as `foo.wares.commonsware.com`.

A <domain-config> element can have a <trust-anchors> element, listing what certificates to use to validate SSL connections made to this domain. Those certificates are identified by <certificate> elements, usually pointing to raw resources that are the PEM or DER files for those certificate authorities. In this case, we point to the addtrustexternalcaroot resource.

To teach Android that you have this network security configuration that you wish to apply, you will need add an android:networkSecurityConfig attribute (for the native Android 7.0 code) and perhaps a <meta-data> element to the <application> element of your manifest (for the CWAC-NetSecurity backport):

```
<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:networkSecurityConfig="@xml/${networkSecurityConfig}">
  <activity
    android:name="DownloaderDemo"
    android:label="@string/app_name">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />

      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>

  <service
    android:name="Downloader"
    android:permission="android.permission.BIND_JOB_SERVICE" />

  <provider
    android:name="LegacyCompatFileProvider"
    android:authorities="${applicationId}.provider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
      android:name="android.support.FILE_PROVIDER_PATHS"
      android:resource="@xml/provider_paths" />
  </provider>

  <meta-data
    android:name="android.security.net.config"
    android:resource="@xml/${networkSecurityConfig}" />
</application>
```

(from Internet/CA/app/src/main/AndroidManifest.xml)

In this case, the resource value used in both places is not a simple XML resource name, like `@xml/network_comodo`, though that will be what most apps will use. This sample application has different product flavors for applying different network security configurations, configured in `build.gradle`. Those product flavors use `manifestPlaceholders` to indicate which XML resource to apply for that flavor:

```
apply plugin: 'com.android.application'

def WARES='"https://wares.commonsware.com/excerpt-7p0.pdf"'
def SELFSIGNED='"https://scrap.commonsware.com:3001/excerpt-7p0.pdf"'

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 17
        targetSdkVersion 27
    }

    flavorDimensions "default"

    productFlavors {
        comodo {
            dimension "default"
            resValue "string", "app_name", "CA Validation Demo"
            applicationId "com.commonsware.android.downloader.ca.comodo"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_comodo']
            buildConfigField "String", "URL", WARES
        }
        verisign {
            dimension "default"
            resValue "string", "app_name", "Invalid CA Validation Demo"
            applicationId "com.commonsware.android.downloader.ca.verisign"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_verisign']
            buildConfigField "String", "URL", WARES
        }
        system {
            dimension "default"
            resValue "string", "app_name", "System CA Validation Demo"
            applicationId "com.commonsware.android.downloader.ca.system"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_verisign_system']
            buildConfigField "String", "URL", WARES
        }
```

```
        pin {
            dimension "default"
            resValue "string", "app_name", "Cert Pin Demo"
            applicationId "com.commonsware.android.downloader.ca.pin"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_pin']
            buildConfigField "String", "URL", WARES
        }
        invalidPin {
            dimension "default"
            resValue "string", "app_name", "Cert Pin Demo"
            applicationId "com.commonsware.android.downloader.ca.invalidpin"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_invalid_pin']
            buildConfigField "String", "URL", WARES
        }
        selfSigned {
            dimension "default"
            resValue "string", "app_name", "Self-Signed Demo"
            applicationId "com.commonsware.android.downloader.ca.ss"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_selfsigned']
            buildConfigField "String", "URL", SELFSIGNED
        }
        override {
            dimension "default"
            resValue "string", "app_name", "Debug Override Demo"
            applicationId "com.commonsware.android.downloader.ca.debug"
            manifestPlaceholders=
                    [networkSecurityConfig: 'network_override']
            buildConfigField "String", "URL", SELFSIGNED
        }
    }
}

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'com.android.support:support-v13:27.0.2'
    implementation 'com.commonsware.cwac:provider:0.5.3'
    implementation 'com.commonsware.cwac:netsecurity:0.4.4'
}
```

(from Internet/CA/app/build.gradle)

The CommonsWare Warescription Web site, at the time of this writing, uses an SSL certificate backed by Comodo. Running the `comodoDebug` build variant should successfully download the PDF file, as the SSL certificate will be validated properly. However, running the `verisignDebug` build variant will fail the SSL validation and crash:

```
03-22 12:51:01.662 27356-27418/com.commonsware.android.downloader.ca E/Exception
downloading file
javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException:
Trust anchor for certification path not found.
   at
com.android.org.conscrypt.OpenSSLSocketImpl.startHandshake(OpenSSLSocketImpl.java:339)
   at com.android.okhttp.Connection.connectTls(Connection.java:235)
   at com.android.okhttp.Connection.connectSocket(Connection.java:199)
   at com.android.okhttp.Connection.connect(Connection.java:172)
   at com.android.okhttp.Connection.connectAndSetOwner(Connection.java:367)
   at com.android.okhttp.OkHttpClient$1.connectAndSetOwner(OkHttpClient.java:130)
   at com.android.okhttp.internal.http.HttpEngine.connect(HttpEngine.java:329)
   at com.android.okhttp.internal.http.HttpEngine.sendRequest(HttpEngine.java:246)
   at
com.android.okhttp.internal.huc.HttpURLConnectionImpl.execute(HttpURLConnectionImpl.java:457)
   at
com.android.okhttp.internal.huc.HttpURLConnectionImpl.getResponse(HttpURLConnectionImpl.java:405)
   at
com.android.okhttp.internal.huc.HttpURLConnectionImpl.getHeaders(HttpURLConnectionImpl.java:162)
   at
com.android.okhttp.internal.huc.HttpURLConnectionImpl.getHeaderField(HttpURLConnectionImpl.java:206)
   at
com.android.okhttp.internal.huc.DelegatingHttpsURLConnection.getHeaderField(DelegatingHttpsURLConnectio
   at
com.android.okhttp.internal.huc.HttpsURLConnectionImpl.getHeaderField(HttpsURLConnectionImpl.java)
   at com.commonsware.android.downloader.Downloader.onHandleIntent(Downloader.java:70)
```

If you have multiple certificate authorities that you wish to support, you can have multiple `<certificate>` elements, or a `<certificate>` element pointing to a file with multiple PEM or DER entries.

Unusual Certificate Authorities

Perhaps your organization runs its own certificate authority (e.g., for internal servers). Or perhaps your organization is using a regular certificate authority, but one that is too new to be recognized by Android. You could cover the unexpected certificate authority by using the `<certificate>` elements shown above.

But, what happens if you want to support something custom *and* regular certificate authorities as well?

In that case, there is a special `<certificate>` element that you can add:

```
<certificates src="system"/>
```

The value `system`, instead of a reference to a raw resource, indicates that the default system set of certificate authorities should be considered to be valid.

The `systemDebug` build variant uses a different network security configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="false" usesCleartextTraffic="false">wares.commonsware.com</domain>
    <trust-anchors>
      <certificates src="@raw/verisign_class3" />
      <certificates src="system" />
    </trust-anchors>
  </domain-config>
</network-security-config>
```

(from Internet/CA/app/src/main/res/xml/network_verisign_system.xml)

Here, first, we pull in Verisign's root certificate. If that were all we had (as you can see in the `network_verisign.xml` resource file), an attempt to download something from `wares.commonsware.com` would fail, as that site uses a Comodo certificate, not a Verisign one. However, we *also* have the `system` set of certificate authorities. Since Comodo is a major certificate authority, it is included in Android's default set, and so our download should succeed.

Pinning the Certificate

Perhaps even supporting any CA's certificates will be too much of a risk for you and your users. For example, perhaps your site's certificate is from a certificate authority that has issued fraudulent credentials in the past, and so you fear that your users might still be at risk of a MITM attack.

You can *really* narrow things down by pinning your app to your specific certificate. Then, only that one certificate will be accepted, not others that might be issued, for your domain, by your certificate authority, either through social engineering, nation-state duress, or whatever.

To do this, you will use a `<pin-set>` element, instead of a `<certificate>` element, in your network security configuration, as seen in the `network_pin` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="false" usesCleartextTraffic="false">wares.commonsware.com</domain>
    <pin-set expiration="2020-05-01">
      <pin digest="SHA-256">sF1A3ez70l81aUjLwU6KiAMmOyPNFQDueJH+4YDWppo=</pin>
    </pin-set>
  </domain-config>
</network-security-config>
```

(from Internet/CA/app/src/main/res/xml/network_pin.xml)

The `<pin-set>` element can include one or more `<pin>` elements, each of which has a `digest` attribute and a value. The `digest` value has to be `SHA-256` at the present time, though perhaps other hash algorithms will be supported in the future. The value of the `<pin>` element is the base64-encoded SHA-256 hash of the `SubjectPublicKeyInfo` field of the X509 certificate of the server.

To generate that value, you will need to use a tool like `openssl`. Given a PEM file named `server.crt`, you can generate the hash for that server using the following command:

```
openssl x509 -in server.crt -pubkey -noout | openssl pkey -pubin -outform der |
openssl dgst -sha256 -binary | openssl enc -base64
```

(NOTE: this should appear all on one line but will be word-wrapped to the size of the book page)

The `<pin-set>` element can also have an `expiration` attribute, with a date in `yyyy-MM-dd` format. Prior to this date, the SSL certificate of the server must match one of the pins. On or after this date, the pins are ignored. For example, you might choose a date that is a bit before the date when the SSL certificate itself will expire. This has the benefit of allowing the app to work even if you fail to update the app and supply a new pin for a new SSL certificate, or if you do update that app but the user does not install the update in time. On the other hand, manually altering the device date and time can bypass your pin.

This behavior — pin expiration allowing formerly-blocked access — is a bit unusual. Typically, with security, we "fail closed", meaning that once something has expired, no access is allowed. Instead, `<pin-set>` specifically "fails open", meaning that once it expires, security is weakened. In this case, Google elected to focus on utility over security.

Self-Signed Certificates

As Moxie Marlinspike points out, one way to avoid having your app be the victim of a man-in-the-middle attack due to a hijacked certificate authority is to simply *not use a certificate authority*.

Certificate authorities are designed for use by general-purpose clients (e.g., Web browsers) hitting general-purpose servers (e.g., Web servers). In the case where you control both the client *and* the server, you don't need a certificate authority. You merely need to have a self-signed certificate that both ends know about.

This works well if the Web server is solely functioning as a Web service to deliver data to your Android app, or perhaps other native apps on other platforms for which you can also support self-signed certificates. Depending upon your server's capabilities, you might be able to arrange to have the same server-side application logic be available both from a self-signed certificate on one domain (for use with apps) and from a CA-rooted certificate for another domain (for use with Web browsers).

However, it is very possible that the staff who manage the servers will reject the notion of using a self-signed certificate, perhaps in an effort to minimize the complexity of supporting multiple SSL paths (for browsers and apps). Or, you may not control the server well enough to go with a self-signed certificate, such as if you are using a cloud computing provider.

However, if self-signed certificates are an option for you, the network security configuration code makes them simple to integrate.

You can use the PEM or DER file from your self-signed certificate much as you would one from a certificate authority: put in `res/raw/` and set up your network security configuration XML to match:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="false" usesCleartextTraffic="false">scrap.commonsware.com</domain>
    <trust-anchors>
      <certificates src="@raw/example" />
    </trust-anchors>
  </domain-config>
</network-security-config>
```

(from Internet/CA/app/src/main/res/xml/network_selfsigned.xml)

This is from the `selfSigned` product flavor. Note that it will not work on your development machine, as you do not have a Web server with a self-signed SSL certificate at `scrap.commonsware.com`. However, this shows the basic setup, as being the same as before.

This site has instructions for setting up a self-signed certificate. The CRT file that is created (e.g., `example.crt`) is what you would put in your app.

Self-Signed Certificates for Debug Builds

If you are only using a self-signed certificate for debuggable builds (e.g., `debug` build type), you can use the `<debug-overrides>` XML element in your network security configuration. This adds your self-signed certificates to the roster of trust anchors, but only for debuggable builds. For non-debuggable builds (e.g., `release` build type), your self-signed SSL certificate will be ignored.

You can see this in the `network_override.xml` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <debug-overrides>
    <trust-anchors>
      <certificates src="@raw/example"/>
    </trust-anchors>
  </debug-overrides>
</network-security-config>
```

(from Internet/CA/app/src/main/res/xml/network_override.xml)

This is for the `override` product flavor which, like `selfSigned`, will not work for you, as you will not have a Web server using that SSL certificate.

Blocking Cleartext Traffic

For a domain, or perhaps for everything in your app, you might want to ensure that you always are using SSL... even to the point of being willing to crash your app if you are *not* using SSL. While this is an extreme measure, some apps have those sorts of security requirements.

The network security configuration subsystem supports a `cleartextTrafficPermitted` attribute on `<base-config>` and `<domain-config>`:

```
<base-config cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

If set to `false`, this means that you want to block all "cleartext" (non-SSL) traffic for the scope of that element.

The native implementation of network security configuration supports this flag for most Internet communications. Notably, `WebView` does not support it.

The CWAC-NetSecurity backport, if you are using the OkHttp3 integration, attempts to honor this, by checking the scheme for requests. If you make an `http` request, but you have `cleartextTrafficPermitted="false"` for the appropriate scope (e.g., for the domain in the URL), the request is rejected. However, this is not quite as strong as the native implementation, and it certainly does not affect anything other than the OkHttp3 integration.

On Android 6.0, you have another option of enabling this same sort of check. You also have a way to have `StrictMode` validate cleartext traffic on Android 6.0+.

Supporting User-Added Certificates

The native Android 7.0+ network security configuration subsystem not only allows you to use `<certificates src="system" />` to say "we also allow any standard certificate authorities here", but also `<certificates src="user" />`. This indicates that certificate authorities added by the user, through Settings, should be honored as well. By default, for apps with `targetSdkVersion` set to 24 or higher, user-added certificates are ignored unless `<certificates src="user" />` is included in a network security configuration.

Such user-added certificate authorities are a bit controversial in Android app development. On the one hand, they allow users to add support for unrecognized authorities, in case Android is slow to adopt them, and without apps having to do anything. On the other hand, those user-added certificate authorities are global in scope, rather than being tied to specific domains.

Note that this feature is not available in the CWAC-NetSecurity backport. `<certificates src="user" />` is ignored. User-added certificate authorities are lumped in with the system-defined certificate authorities, so if you have

`<certificates src="system" />`, you will get certificate authorities from both sources.

Other SSL Strengthening Techniques

Not everything that one can do to improve SSL security is covered by either the native network security configuration implementation or the CWAC-NetSecurity backport. Here are some other possibilities to consider.

Certificate Memorizing

If your app needs to connect to arbitrary SSL servers — perhaps ones configured by the user (e.g., email client) or are intrinsic to the app's usage (e.g., URLs in a Web browser) — detecting man-in-the-middle attacks boils down to proper SSL certificate validation... and praying for no hacked CA certificates.

However, one way to incrementally improve security is to use certificate memorizing. With this technique, each time you see a certificate that you have not seen before, or perhaps a different certificate for a site visited previously, you ask the user to confirm that it is OK to proceed.

The idea here is that even if we cannot tell, absolutely, whether a given certificate is genuine or from an attacker, we can detect *differences in certificates over time*. So, if the user has been seeing certificate A, and now all of a sudden receives certificate B instead, there are two main possibilities:

1. The HTTPS server changed certificates for legitimate reasons
2. An attacker is providing an alternative certificate

So, what we do is check certificates against a roster that the user has approved before. If the newly-received certificate is not in that roster, we fail the HTTPS request, but raise a custom exception so that your code can detect this case and ask the user for approval to proceed.

Technically savvy users may be able to deduce whether the certificate is indeed genuine; slightly less-savvy users might simply contact the site to see if this is expected behavior. The downside is that technically unsophisticated users might be baffled by the question of whether or not they should accept the certificate and may take their confusion out on you, the developer of the app that is asking the question.

There is a standalone implementation of a `MemorizingTrustManager` that you could consider using. It has been around for a few years, with a slow-but-steady set of updates.

However, that library handles asking the user for acceptance of the certificates for you, rather than raising some event that your app can handle itself. In order to tailor the UI, you would need to modify the library itself.

Moreover, the library attempts to handle this UI *while your SSL request is in process*, by blocking the background thread upon which you are making the HTTPS request. A side-effect of this is that `MemorizingTrustManager` has some fairly unpleasant code for trying to block this thread while interacting with the user on the main application thread. And, if the user takes too long, your request to the server may time out anyway.

Requiring Encryption, Android 6.0 Style

Android 6.0 supports an `usesCleartextTraffic` attribute on the `<application>` element in the manifest. This works like the `cleartextTrafficPermitted` option in the network security configuration subsystem. If this is set to `false`, you are saying that your app not only should be using SSL for everything, but that you expressly want to crash the app in case you wind up *not* using SSL.

If you try to perform plain HTTP requests on Android 6.0 with `usesCleartextTraffic` set to `false`, you will crash when you attempt to download the file, with a stack trace akin to:

```
06-19 08:03:46.325    6420-6478/com.commonsware.android.downloader E/
com.commonsware.android.downloader.Downloader: Exception in download
    java.net.UnknownServiceException: CLEARTEXT communication not supported: []
            at com.android.okhttp.Connection.connect(Connection.java:149)
            at com.android.okhttp.Connection.connectAndSetOwner(Connection.java:185)
            .
            .
            .
```

What is really going on "under the covers" is that this attribute sets a flag that HTTP client APIs can check, electing to fail a request if the flag says that SSL is required and the request's URL does not have the `https` scheme. Android's built-in HTTP clients should support this flag, but third-party HTTP stacks that manage their own socket connections may not. Also note that `WebView` does not honor `usesCleartextTraffic`.

Watching for Encryption

The downside of `usesCleartextTraffic` is that it is "all or nothing" and always terminates your process. The same thing holds true for using `cleartextTrafficPermitted` with the network security configuration in the `<base-config>` element. That is wonderful in situations where SSL is crucial. It is less wonderful if your app crashes in production in situations where SSL would be a really good idea but is unavailable for whatever reason.

`StrictMode` on API Level 23+ devices supports a way to be warned if your app performs unencrypted network operations, via a `detectCleartextNetwork()` method on `StrictMode.VmPolicy.Builder`. You can configure this, and suitable penalties, alongside the rest of your `StrictMode` setup. This can include doing different things for `debug` versus `release` builds, for example. So, in a `debug` build, you might choose `penaltyDeath()` to crash the process, while in a `release` build, you settle for `penaltyLog()` or something else less drastic.

If you are using a build server, you could set it up to watch for `StrictMode` Logcat messages coming from your test suite to find out about these accesses.

Advanced Uses of CWAC-NetSecurity

Adding a couple of lines of Java code, along with the dependency, is all that you need to use CWAC-NetSecurity to gain the benefits of the backport of the network security configuration subsystem. However, CWAC-NetSecurity offers a few more features that may be of use to you.

Using Alternative Network Security Configuration XML

`withManifestConfig()` on `TrustManagerBuilder` uses the resource that you declare in your manifest as the network security configuration to apply. However, that is fairly inflexible, as you can only define this in the manifest once. Also, `withManifestConfig()` performs the version check to only apply the backport on pre-7.0 devices.

You can also use `withConfig()`, where you provide a `Context` and the resource ID of the XML resource to use for the network security configuration. This is useful for cases where:

- You want to always use the backport, for consistent behavior across OS versions
- You want to use different configurations in different settings for the same APK

For example, the test suites use `withConfig()`, as otherwise we would need dozens of separate manifests.

Using the Backport Directly

You do not have to use `TrustManagerBuilder` to use the network security configuration backport. If you wish to use it directly:

- Create an instance of `ApplicationConfig`, passing in a `ConfigSource` implementation that indicates where the configuration should be pulled from. Two likely `ConfigSource` implementations are `ManifestConfigSource` (to use the one defined in the manifest) and `XmlConfigSource` (to use one defined in an arbitrary XML resource).
- Call `getTrustManager()` on the `ApplicationConfig` to get a `TrustManager` that will implement the requested configuration.
- Add that `TrustManager` to your HTTP client via whatever API that client offers for such things. In many cases, that will be by configuring an `SSLContext` to use the `TrustManager`, then using the `SSLContext` (or an `SSLSocketFactory` created by the `SSLContext`) with your HTTP client.

Integrating with Other HTTP Client Libraries

If you want to integrate `TrustManagerBuilder` and the network security configuration backport with some other HTTP client API, start by reviewing the `OkHttp3Integrator` class in the `netsecurity-okhttp3` library. This will give you an idea of what is required and how easy it will be to replicate this class for your particular HTTP client API.

Adding the TrustManager

Calling `build()` on the `TrustManagerBuilder` gives you a `CompositeTrustManager`, set up to implement your desired network security configuration. You will need to add that to your HTTP client by one means or another. If `size()` on the `CompositeTrustManager` returns 0, though, you can skip it, as it means that there are

no rules to be applied (e.g., you used `withManifestConfig()`, and your app is running on an Android 7.0+ device).

So, you might have code that looks like this, where `tmb` is a configured `TrustManagerBuilder`:

```
CompositeTrustManager trustManager=tmb.build();

if (trustManager.size()>0) {
  SSLContext ssl=SSLContext.getInstance("TLS");
  X509Interceptor interceptor=new X509Interceptor(trustManager, tmb);

  ssl.init(null, new TrustManager[]{trustManager}, null);

  // apply the SSLContext or ssl.getSocketFactory() to your HTTP client
}
```

Handling Cleartext

You can call `isCleartextTrafficPermitted()` on the `CompositeTrustManager` to determine if cleartext traffic should be supported. This takes the domain name of the Web server you are going to be communicating with and returns a simple `boolean`. If `isCleartextTrafficPermitted()` returns `false`, you will need to examine the scheme of the URL and accept or reject the HTTP operation accordingly.

If you fail to do this, then cleartext traffic will be allowed in all cases, akin to the stock `HttpURLConnection` integration.

Handling Redirects

If your HTTP client automatically traverses server-side redirects (making the HTTP request for the redirected-to URL), you will need to handle the cleartext check and the `setHost()` call on every step of the redirection, not just your initial request. In the case of OkHttp3, this is accomplished via their interceptor framework.

Debugging Certificate Chains

You can call `withCertChainListener()` on `TrustManagerBuilder`, providing an implementation of `CertChainListener`. Your listener will be called with `onChain()` each time a certificate chain is encountered. In `onChain()`, you can inspect the certificates, dump their contents to Logcat, or whatever you wish to do.

This is designed for use in development. For example, when writing the `demo/` app, the author used a `CertChainListener` to log what HTTP requests were being made, what domains those were for, and what root certificates are being used. This in turn led to creating the network security configuration that matched.

However, logging certificate chains on a production device may result in security issues. Please only use `CertChainListener` in debug builds.

NetCipher

The Guardian Project has released an Android library project called NetCipher — formerly known as OnionKit — designed to help boost Internet security for Android applications.

In particular, NetCipher helps your application integrate with Orbot, a Tor proxy. Tor ("The Onion Router") is designed to help with anonymity, having your Internet requests go through a series of Tor routers before actually connecting to your targeted server through some Tor endpoint. Tor is used for everything from mitigating Web site tracking to helping dissidents bypass national firewalls. NetCipher helps your app:

- Detect if Orbot is installed, and help the user install it if it is not
- Detect if Orbot is running, and help you start it if it is not
- Make HTTP requests by means of Orbot instead of directly over the Internet

There is a dedicated chapter on NetCipher, if you have interest in this technology.

NetCipher

NetCipher is a library from the Guardian Project to improve the privacy and security of HTTP network communications. In particular, it makes it easier for your app to integrate with Orbot, an Android proxy server that forwards HTTP requests via Tor.

This chapter covers:

- An introduction to Tor, Orbot, and NetCipher
- An explanation of how to use a fairly simple API layered atop NetCipher to add its functionality to your app

Prerequisites

This chapter assumes that you have read the core chapters of the book, particularly the one on Internet access. Having read the chapter on SSL is also a very good idea.

Network Security's Got Onions

Maintaining privacy and security on the Internet, in the face of so-called "advanced persistent threats", is a continuous challenge facing many people, particularly those under threats from hostile forces, ranging from organized crime syndicates to your average rampaging warlord. Tor was created to help deal with this sort of problem; Orbot was created to extend Tor to Android.

A Quick Primer on Tor

Originally named The Onion Router, Tor was created by researchers in the US Naval Research Laboratory back in the mid-1990's, with an eye towards protecting US

intelligence communications. In 2006, the technology spun out into an independent non-profit organization, which has continued to improve upon the core Tor software and expand the reach of Tor. Through packages like the Tor Browser Bundle, it is fairly easy for at-risk people to start using Tor to help shroud their communications.

Without getting into the full technical details of Tor — which are *well* beyond the scope of this chapter — Tor basically works by routing a request through a series of relay servers, through a process known as onion routing. Requests are secured through layers of encryption, to keep any two connected relays from knowing the full details of the communications. Some relays serve as "exit nodes", for requests being made of ordinary Web servers. Certain servers — Tor hidden services — are only reachable through Tor; requests made of these servers never leave the Tor network.

Of course, technology like Tor is agnostic in terms of its users and usages, and there have been plenty of examples of people using Tor for illicit purposes, such as the Silk Road. This has a tendency to obscure Tor's benefits to people who need to remain somewhat hidden online, whether from stalkers or other harassers or from the security forces of dictatorships.

Introducing Orbot

The entry path into Tor is usually via some sort of proxy server, that a regular Internet client can connect to. Orbot is one such proxy server, that runs on Android. Apps can use Orbot's HTTP or SOCKS proxies to route requests; those requests will then wind up traversing the Tor network to the end site, whether that site is on the public Internet (reached from a Tor exit node) or a Tor hidden service.

By default, Orbot is limited to `localhost` use, meaning that it does not have open ports that can be reached from other devices on the local WiFi LAN segment (or some subnet of the mobile carrier, if not on WiFi). For an Android app on the same device, this is not a problem, and it in fact simplifies things a fair bit, as there is no guesswork as to what the IP address should be for the proxy. As we will see, though, finding out exactly how to connect to Orbot is a bit tricky, though with some helper code it is not too bad.

What NetCipher Provides

While we know that Orbot will be listening on `localhost`, we do not necessarily know the port that it is using for its HTTP proxy. Partly, that is because the user

might configure it manually. Partly, that is because there are occasional conflicts with Orbot's default port.

Hence, NetCipher contains some code that will help you find out:

- Is Orbot installed? (and, if not, help get it installed)
- Is Orbot running? (and, if not, help get it running)
- What port is used for the HTTP proxy?

The NetCipher HTTP Integration APIs

NetCipher offers two levels of API for integration. This chapter focuses on the newer of those, a suite that offers simple plug-and-play integration with popular HTTP client APIs: `HttpURLConnection`, OkHttp, Volley, and Apache's HttpClient. This focus stems from two main reasons:

- These integration APIs are much simpler to use
- The author of this book wrote those APIs, and so is biased as to how simple they are to use

The `Internet/HTTPStacks` sample application demonstrates all four of the HTTP integration APIs. Each of the four is based on the Stack Overflow sample app from the chapter on Internet access, with NetCipher integration added in.

There are a few simple steps for adding in NetCipher integration: choosing your HTTP stack, adding the dependencies, setting up `OrbotHelper`, and then using a secure connection.

Choose an HTTP Stack

As noted above, NetCipher offers integration APIs for four major HTTP client implementations (a.k.a., "HTTP stacks"):

- `HttpURLConnection`
- OkHttp3
- Apache's independent HttpClient package
- Volley

`HttpURLConnection` support is part of the core NetCipher library (`info.guardianproject.netcipher:netcipher`), as `HttpURLConnection` is part of standard Java and Android. The other three HTTP stacks have separate libraries:

HTTP Stack	NetCipher Artifact
OkHttp3	`info.guardianproject.netcipher:netcipher-okhttp3`
HttpClient	`info.guardianproject.netcipher:netcipher-httpclient`
Volley	`info.guardianproject.netcipher:netcipher-volley`

Add the Dependencies

Unfortunately, some packaging issues with the `2.0.0-alpha1` edition of NetCipher, adding the dependencies is more complicated than it needs to be. Your project needs to have dependencies on:

- `info.guardianproject.netcipher:netcipher:2.0.0-alpha1`, for the core of NetCipher
- the NetCipher artifact specifically for your HTTP stack, if you are using something other than `HttpURLConnection`
- the artifact for the HTTP client API itself
- any other artifacts that your project needs for other reasons

So, for example, the `okhttp3` module in the `HTTPStacks` project is a sample app that uses OkHttp 3.x for its HTTP client API. It needs three artifacts in its `dependencies` closure to pull in OkHttp and NetCipher's support for OkHttp:

```
    compile 'info.guardianproject.netcipher:netcipher:2.0.0-alpha1'
    compile 'info.guardianproject.netcipher:netcipher-okhttp3:2.0.0-alpha1'
    compile 'com.squareup.okhttp3:okhttp:3.8.0'
```

(from Internet/HTTPStacks/okhttp3/build.gradle)

Volley integration has been tested with `com.android.volley:volley:1.0.0`, while the HttpClient integration work with the `cz.msebera.android:httpclient:4.4.1.2` independent repackaging of Apache HttpClient for Android.

Set up OrbotHelper

`OrbotHelper` is a singleton that manages a lot of the asynchronous communication between your app and Orbot. It is designed to be initialized fairly early on in your

app's lifecycle. One likely candidate is to have a custom `Application` subclass, where you override `onCreate()` and set up `OrbotHelper`.

All of the sample apps do this in a custom `SampleApplication` class:

```
package com.commonsware.android.http;

import android.app.Application;
import com.squareup.leakcanary.LeakCanary;
import info.guardianproject.netcipher.proxy.OrbotHelper;

public class SampleApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    LeakCanary.install(this);
    OrbotHelper.get(this).init();
  }
}
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/SampleApplication.java)

This custom `Application` also sets up LeakCanary.

`SampleApplication` is then tied into the app via the `android:name` attribute on the `<application>` element in the manifest:

```
  <application
    android:name=".SampleApplication"
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
```

(from Internet/HTTPStacks/okhttp3/src/main/AndroidManifest.xml)

Choose and Create a Builder

Each module defines a corresponding builder class that can be used to configure NetCipher for use with that stack, with names based on the classes used with those HTTP stacks:

HTTP Stack	Builder Class
`HttpURLConnection`	`StrongConnectionBuilder`

HTTP Stack	Builder Class
OkHttp3	`StrongOkHttpClientBuilder`
HttpClient	`StrongHttpClientBuilder`
Volley	`StrongVolleyQueueBuilder`

You will need an instance of your chosen builder class. The simplest way to do that is to call the `forMaxSecurity()` `static` method on the builder class. `forMaxSecurity()` takes a `Context` as a parameter, though it only holds onto the `Application` singleton internally, so any `Context` is safe. `forMaxSecurity()` returns a builder configured for the best protection that NetCipher can offer.

Get a Connection

Then, call `build()` on the builder object. It will take a `StrongBuilder.Callback` object as a parameter, typed for whatever HTTP stack you chose. So, for example, if you went with `StrongConnectionBuilder`, your callback will be a `StrongBuilder.Callback<HttpURLConnection>`.

HTTP Stack	Builder Class	Connection Class
`HttpURLConnection`	`StrongConnectionBuilder`	`HttpURLConnection`
OkHttp3	`StrongOkHttpClientBuilder`	`OkHttpClient`
HttpClient	`StrongHttpClientBuilder`	`HttpClient`
Volley	`StrongVolleyQueueBuilder`	`RequestQueue`

You will need to implement four methods on that `Callback`:

- `onConnected()` will be passed an instance of your connection class (e.g., an `HttpURLConnection` instance), ready for your use, configured to hook into NetCipher
- `onConnectionException()` will be passed an `IOException`, if one of those occurs while trying to set up your connection
- `onTimeout()` will be called if Orbot is not installed or we could not connect to it within 30 seconds
- `onInvalid()` will be called if the Tor connection is established but is deemed to be compromised (more on this later)

Seeing the Builder in Action

Each of the four modules in the sample app (`hurl`, `httpclient`, `okhttp3`, and `volley`) have a similar `MainActivity` implementation, one that populates a

`ListView` with the latest Stack Overflow Android questions. The difference in which HTTP stack the sample uses.

For example, the `okhttp3` module, in `onCreate()` of its `MainActivity`, uses `StrongOkHttpClientBuilder`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  try {
    StrongOkHttpClientBuilder
      .forMaxSecurity(this)
      .withTorValidation()
      .build(this);
  }
  catch (Exception e) {
    Toast
      .makeText(this, R.string.msg_crash, Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(),
      "Exception loading SO questions", e);
    finish();
  }
}
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/MainActivity.java)

Here, we use `forMaxSecurity()` to create the `StrongOkHttpClientBuilder`, then configure it further with `withTorValidation()`. This requests that we do a test HTTP request to a Tor status URL to confirm that our request has indeed gone over Tor.

Note that `StrongConnectionBuilder` — for use with `HttpURLConnection` – also requires that you call `connectTo()`, before `build()`, to indicate the specific URL for which you want an `HttpURLConnection`. This is unique among the builders. These sorts of per-builder differences are discussed later in this chapter.

`build()` is passed `this`, referencing `MainActivity` itself, which is implementing the `StrongBuilder.Callback` interface:

```
public class MainActivity extends ListActivity implements
  StrongBuilder.Callback<OkHttpClient> {
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/MainActivity.java)

That `Callback` is tied to the particular type of connection we are creating. We are using OkHttp3 and `StrongOkHttpClientBuilder`, so we are creating an `OkHttpClient` connection.

Our `onConnected()` method for that `Callback` gets the `OkHttpClient` and makes an HTTP request using it:

```
@Override
public void onConnected(final OkHttpClient client) {
  new Thread() {
    @Override
    public void run() {
      try {
        Request request=new Request.Builder().url(SO_URL).build();
        Response response=client.newCall(request).execute();

        final SOQuestions result=
          new Gson().fromJson(response.body().charStream(), SOQuestions.class);

        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            setListAdapter(new ItemsAdapter(result.items));
          }
        });
      }
      catch (IOException e) {
        onConnectionException(e);
      }
    }
  }.start();
}
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/MainActivity.java)

`SO_URL`, passed into `url()`, is a Web service request URL from the Stack Exchange API, looking for Stack Overflow questions tagged with the `android` tag:

```
String SO_URL=
  "https://api.stackexchange.com/2.1/questions?"
    + "order=desc&sort=creation&site=stackoverflow&tagged=android";
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/MainActivity.java)

Note that `onConnected()` will be called on the main application thread, so you will need to get your connection over to whatever background thread will be doing your work. In this case, we create a background thread right here to retrieve the JSON, parse it, and use `runOnUiThread()` to update the `ListActivity` with an `ItemsAdapter` to show the parsed Stack Overflow questions:

```
class ItemsAdapter extends ArrayAdapter<Item> {
  ItemsAdapter(List<Item> items) {
    super(MainActivity.this,
      android.R.layout.simple_list_item_1, items);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    TextView title=(TextView)row.findViewById(android.R.id.text1);

    title.setText(Html.fromHtml(getItem(position).title));

    return(row);
  }
}
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/MainActivity.java)

The other three methods that we need to implement for our `Callback` are for error conditions: `onConnectionException()`, `onTimeout()`, and `onInvalid()`:

```
@Override
public void onConnectionException(Exception e) {
  Log.e(getClass().getSimpleName(),
    "Exception loading SO questions", e);

  runOnUiThread(new Runnable() {
    @Override
    public void run() {
      Toast
        .makeText(MainActivity.this, R.string.msg_crash,
          Toast.LENGTH_LONG)
        .show();
      finish();
    }
  });
}

@Override
public void onTimeout() {
  runOnUiThread(new Runnable() {
    @Override
    public void run() {
      Toast
        .makeText(MainActivity.this, R.string.msg_timeout,
          Toast.LENGTH_LONG)
        .show();
```

```
      finish();
    }
  });
}

@Override
public void onInvalid() {
  runOnUiThread(new Runnable() {
    @Override
    public void run() {
      Toast
        .makeText(MainActivity.this, R.string.msg_invalid,
          Toast.LENGTH_LONG)
        .show();
      finish();
    }
  });
}
```

(from Internet/HTTPStacks/okhttp3/src/main/java/com/commonsware/android/http/MainActivity.java)

Other than initializing `OrbotHelper`, setting up the builder, and implementing `StrongBuilder.Callback` somewhere to handle the results, the rest of the code is tied to application logic, not NetCipher itself.

The Rest of the Builder API

The API shown above for getting a NetCipher-secured connection via your favorite HTTP stack is designed for ease of use. However, as shown, it is not very flexible.

The rest of the builder API offers that flexibility, at the cost of some additional code.

Common Configuration Methods

The `StrongBuilder` interface defines the common public API for all four of the builder classes:

```
/*
 * Copyright (c) 2016 CommonsWare, LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
```

```
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package info.guardianproject.netcipher.client;

import android.content.Intent;
import java.io.IOException;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;
import javax.net.ssl.TrustManager;

public interface StrongBuilder<T extends StrongBuilder, C> {
  /**
   * Callback to get a connection handed to you for use,
   * already set up for NetCipher.
   *
   * @param <C> the type of connection created by this builder
   */
  interface Callback<C> {
    /**
     * Called when the NetCipher-enhanced connection is ready
     * for use.
     *
     * @param connection the connection
     */
    void onConnected(C connection);

    /**
     * Called if we tried to connect through to Orbot but failed
     * for some reason
     *
     * @param e the reason
     */
    void onConnectionException(Exception e);

    /**
     * Called if our attempt to get a status from Orbot failed
     * after a defined period of time. See statusTimeout() on
```

```
     * OrbotInitializer.
     */
    void onTimeout();

    /**
     * Called if you requested validation that we are connecting
     * through Tor, and while we were able to connect to Orbot, that
     * validation failed.
     */
    void onInvalid();
}

/**
 * Call this to configure the Tor proxy from the results
 * returned by Orbot, using the best available proxy
 * (SOCKS if possible, else HTTP)
 *
 * @return the builder
 */
T withBestProxy();

/**
 * @return true if this builder supports HTTP proxies, false
 * otherwise
 */
boolean supportsHttpProxy();

/**
 * Call this to configure the Tor proxy from the results
 * returned by Orbot, using the HTTP proxy.
 *
 * @return the builder
 */
 T withHttpProxy();

/**
 * @return true if this builder supports SOCKS proxies, false
 * otherwise
 */
 boolean supportsSocksProxy();

/**
 * Call this to configure the Tor proxy from the results
 * returned by Orbot, using the SOCKS proxy.
 *
 * @return the builder
 */
T withSocksProxy();
```

```
/**
 * Applies your own custom TrustManagers, such as for
 * replacing the stock keystore support with a custom
 * keystore.
 *
 * @param trustManagers the TrustManagers to use
 * @return the builder
 */
T withTrustManagers(TrustManager[] trustManagers)
  throws NoSuchAlgorithmException, KeyManagementException;

/**
 * Call this if you want a weaker set of supported ciphers,
 * because you are running into compatibility problems with
 * some server due to a cipher mismatch. The better solution
 * is to fix the server.
 *
 * @return the builder
 */
T withWeakCiphers();

/**
 * Call this if you want the builder to confirm that we are
 * communicating over Tor, by reaching out to a Tor test
 * server and confirming our connection status. By default,
 * this is skipped. Adding this check adds security, but it
 * has the chance of false negatives (e.g., we cannot reach
 * that Tor server for some reason).
 *
 * @return the builder
 */
T withTorValidation();

/**
 * Builds a connection, applying the configuration already
 * specified in the builder.
 *
 * @param status status Intent from OrbotInitializer
 * @return the connection
 * @throws IOException
 */
C build(Intent status) throws Exception;

/**
 * Asynchronous version of build(), one that uses OrbotInitializer
 * internally to get the status and checks the validity of the Tor
 * connection (if requested). Note that your callback methods may
```

```
   * be invoked on any thread; do not assume that they will be called
   * on any particular thread.
   *
   * @param callback Callback to get a connection handed to you
   *                 for use, already set up for NetCipher
   */
  void build(Callback<C> callback);
}
```

`withTorValidation()`, `build()`, and the `Callback` nested interface were covered earlier in this chapter, but the others offer finer-grained configuration options.

Five of the methods are tied into choosing what proxy protocol should be used with Orbot.

`forMaxSecurity()`, under the covers, uses `withBestProxy()`, which chooses the best proxy for the situation. Right now, the implementation chooses the SOCKS proxy where that is supported, falling back to the HTTP proxy where it is not.

The `supportsHttpProxy()` and `supportsSocksProxy()` methods indicate whether a given builder supports these proxy types.

The `withHttpProxy()` and `withSocksProxy()` methods tell the builder that you want to use that specific proxy. Use these with care, making sure that the proxy you want is supported. `withBestProxy()` is a far better choice overall.

`withWeakCiphers()` expands the roster of SSL ciphers that NetCipher allows the HTTPS connection to use. Normally, NetCipher tries to avoid ciphers with known security issues. However, that may cause problems with some servers, if NetCipher and the server cannot negotiate a common cipher. `withWeakCiphers()` allows NetCipher to use more ciphers, to perhaps overcome the negotiation problem, with the cost of possibly weaker security.

`withTrustManagers()` allows you to replace the `TrustManager` implementation that NetCipher would use by default with a different one, perhaps one that supports certificate pinning or other SSL strengthening techniques.

Differences Between the Stacks

While each of the builders supports the `StrongBuilder` API, there are some differences between the implementations.

StrongConnectionBuilder

Before calling `build()`, you need to call `connectTo()` to supply the URL (as a `String` or `URL`) that you want to connect to. The other builders give you objects that you can reuse across many requests (e.g., OkHttp3's `OkHttpClient`), but that is not possible with `HttpURLConnection`.

The `hurl` module's `MainActivity` does just that:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  try {
    StrongConnectionBuilder
      .forMaxSecurity(this)
      .withTorValidation()
      .connectTo(SO_URL)
      .build(this);
  }
  catch (Exception e) {
    Toast
      .makeText(this, R.string.msg_crash, Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(),
      "Exception loading SO questions", e);
    finish();
  }
}
```

(from Internet/HTTPStacks/hurl/src/main/java/com/commonsware/android/http/MainActivity.java)

The `onConnected()` method then just uses the fully-configured `HttpURLConnection` object:

```
@Override
public void onConnected(final HttpURLConnection conn) {
  new Thread() {
    @Override
    public void run() {
      try {
        InputStream in=conn.getInputStream();
        BufferedReader reader=
          new BufferedReader(new InputStreamReader(in));

        final SOQuestions result=
```

```
          new Gson().fromJson(reader, SOQuestions.class);

        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            setListAdapter(new ItemsAdapter(result.items));
          }
        });

        reader.close();
      }
      catch (IOException e) {
        onConnectionException(e);
      }
      finally {
        conn.disconnect();
      }
    }
  }.start();
}
```

(from Internet/HTTPStacks/hurl/src/main/java/com/commonsware/android/http/MainActivity.java)

To help make this a bit easier, `StrongConnectionBuilder` supports the copy constructor. You can create a master `StrongConnectionBuilder` with your base configuration, then make a copy, call `connectTo()` on the copy, then call `build()` on the copy, throwing away the copy when you are done.

StrongHttpClientBuilder

The builder for Apache's independent packaging of HttpClient for Android extends Apache's own `HttpClientBuilder`. As a result, you can call all the normal `HttpClientBuilder` methods in addition to calling the `StrongBuilder` methods. The noteworthy exception is that the standard zero-parameter `build()` offered by `HttpClientBuilder` is not supported.

The `httpclient` module's `MainActivity` does not need any HttpClient-specific configuration:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  try {
    StrongHttpClientBuilder
```

```
      .forMaxSecurity(this)
      .withTorValidation()
      .build(this);
  }
  catch (Exception e) {
    Toast
      .makeText(this, R.string.msg_crash, Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(),
      "Exception loading SO questions", e);
    finish();
  }
}
```

(from Internet/HTTPStacks/httpclient/src/main/java/com/commonsware/android/http/MainActivity.java)

The `onConnected()` method then just uses the configured `HttpClient` object:

```
@Override
public void onConnected(final HttpClient client) {
  new Thread() {
    @Override
    public void run() {
      try {
        HttpGet get=new HttpGet(SO_URL);
        String json=client.execute(get, new BasicResponseHandler());

        final SOQuestions result=
          new Gson().fromJson(new StringReader(json),
            SOQuestions.class);

        runOnUiThread(new Runnable() {
          @Override
          public void run() {
            setListAdapter(new ItemsAdapter(result.items));
          }
        });
      }
      catch (IOException e) {
        onConnectionException(e);
      }
    }
  }.start();
}
```

(from Internet/HTTPStacks/httpclient/src/main/java/com/commonsware/android/http/MainActivity.java)

StrongVolleyQueueBuilder

This builder class adheres to the `StrongBuilder` API without any changes, making its use fairly straightforward:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  try {
    StrongVolleyQueueBuilder
      .forMaxSecurity(this)
      .withTorValidation()
      .build(this);
  }
  catch (Exception e) {
    Toast
      .makeText(this, R.string.msg_crash, Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(),
      "Exception loading SO questions", e);
    finish();
  }
}
```

(from Internet/HTTPStacks/volley/src/main/java/com/commonsware/android/http/MainActivity.java)

The `onConnected()` method then just uses the configured `RequestQueue` object:

```
@Override
public void onConnected(final RequestQueue rq) {
  new Thread() {
    @Override
    public void run() {
      final StringRequest stringRequest=
        new StringRequest(StringRequest.Method.GET, SO_URL,
          new Response.Listener<String>() {
            @Override
            public void onResponse(String response) {
              final SOQuestions result=
                new Gson().fromJson(new StringReader(response),
                  SOQuestions.class);

              runOnUiThread(new Runnable() {
                @Override
                public void run() {
                  setListAdapter(new ItemsAdapter(result.items));
```

```
                }
              });
            }
          },
          new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
              Log.e(getClass().getSimpleName(),
                "Exception making Volley request", error);
            }
          });

      rq.add(stringRequest);
    }
  }.start();
}
```

(from Internet/HTTPStacks/volley/src/main/java/com/commonsware/android/http/MainActivity.java)

StrongOkHttpClientBuilder

Note that OkHttp3 does not support SOCKS proxies. Hence, `supportsSocksProxy()` returns `false`, causing `withBestProxy()` to fall back to the HTTP proxy. This is handled for you automatically.

Trail: Advanced Network Topics

Embedding a Web Server

Usually, Android devices are mobile. Usually, servers are not mobile.

However, occasionally, you may have a valid reason to want to have your Android app expose some sort of open TCP/IP port to other apps, the user, or (eek!) the Internet at large. The "eek!" is because allowing foreign devices access to stuff inside a user's device is fraught with security issues, as usually Android devices lack configurable firewalls and the other protection measures associated with production-grade servers.

In this chapter, we will explore some reasons for having such a TCP daemon as part of your app, focusing on the most common scenario: serving Web content from your app. We will then examine more closely one embeddable Web server implementation and how you can use it — carefully – in your Android apps.

Prerequisites

In addition to having read the core chapters of this book, you should have some familiarity with setting up a Web server and a Web application. This chapter is not a primer on these topics, but instead focuses on how to do them in the context of an Android app.

Why a Web Server?

Some people reading this chapter might wonder what role a Web server would ever have being in an Android app. Sure, we *talk* to Web servers all the time, particularly those hosting Web services. But *publishing* a Web server is uncommon, to say the least.

However, “uncommon” does not mean “completely ridiculous”. Even though there are security concerns with having Web servers embedded in Android apps, there are plenty of use cases as well.

Development Uses

One way to mitigate the security issues is to use the Web server only in constrained situations. One common situation is “development”: if the Web server is only used in, say, `debug` builds, we do not have to worry about security concerns affecting ordinary users. This reduces the potential audience of those who might be affected by some attacker.

One popular example of this is Facebook’s Stetho, which extends Chrome DevTools to be able to examine Android apps, including:

- Examining the view hierarchy in much the same was as you view the DOM of a loaded Web page
- Optionally using OkHttp interceptors to monitor network requests in much the same way as you can view network activity by a Web page
- Examine SQLite databases much as you might examine cookies, local storage, or other Web client-side storage mechanisms

Stetho accomplishes this through an embedded HTTP daemon that Chrome DevTools communicates with.

This book’s chapter on custom in-app diagnostic tools will examine how you can use the techniques outlined in this chapter to build your own Stetho-style diagnostics.

Other tools, like Opersys’ Binder Explorer, serve up Web content from a device, but are standalone tools, not designed to be embedded in an app.

Production Uses

Running Web servers on end-user devices is a bit frightening. Not only do normal Android security measures, like permissions, play much of a role, but we lack most of the security infrastructure seen with traditional Web servers.

The counterbalance is that mobile devices rarely have public IP addresses. This limits the scope of potential attackers to those on the same network. Later in this chapter, we will explore various ways of securing these sorts of servers from this limited attacker audience.

Putting the security issues aside for a moment, there is one main reason why one might want to run a Web server on a mobile device: you want other things, outside the device, to talk to your app. There are many possible use cases here, such as:

- Wanting to serve media stored on the device to playback devices, like having Chromecast play a movie stored on a phone
- Wanting to allow users to work with device-resident data from their desktop or notebook computers, via a Web app, instead of having to have that data be synchronized to some third-party Web site
- Exposing Web services to access device-resident data, such as having native programs on desktops or notebooks talk to your app and have access to its data
- Serving content to a small group, such as meeting participants at a neutral site, perhaps from a device more dedicated to that role (e.g., an Android HDMI dongle, as opposed to any one person's phone or tablet)

Introducing AsyncHttpServer

There are a variety of HTTP servers available for Android. Some are standalone programs, such as Opersys' cross-compiled node.js used for their Binder Explorer. Others are embeddable, designed to be used from Android apps. One of the more prominent of these comes from Koushik Dutta's AndroidAsync project.

Among the TCP/IP clients and servers in AndroidAsync is `AsyncHttpServer`. As the name suggests, it is an implementation of an HTTP server, offering a reasonable range of features:

- Pluggable providers of content for particular URL routes, for implementing Web app-style interfaces, including "out of the box" support for serving directories containing files
- WebSockets support, for pushing data to clients in addition to responding to incoming HTTP requests
- Configurable ports (other than the standard Linux/Android limitation that you cannot use ports below 1024, since you do not have superuser privileges)
- SSL support via a configurable `SSLContext`

Embedding a Simple Server

From the standpoint of `AsyncHttpServer`, getting the server going is almost trivial. The `AsyncHttpServer` documentation shows examples like this:

```
AsyncHttpServer server = new AsyncHttpServer();

server.get("/", new HttpServerRequestCallback() {
    @Override
    public void onRequest(AsyncHttpServerRequest request, AsyncHttpServerResponse
response) {
        response.send("Hello!!!");
    }
});

// listen on port 5000
server.listen(5000);
// browsing http://localhost:5000 will return Hello!!!
```

However, there is more to using `AsyncHttpServer` when you start to take into account things like UI controls, foreground services, and the like.

The `WebServer/Simple` sample application demonstrates a fairly minimal complete app that uses `AsyncHttpServer` to serve up some content.

The Dependencies

This app uses three dependencies:

- AndroidAsync, for obvious reasons
- greenrobot's EventBus, so the service hosting the `AsyncHttpServer` can let the UI layer know — if the UI exists — about changes in the state of the server
- `support-v13`, mostly for `NotificationCompat`, used for creating the foreground service

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.koushikdutta.async:androidasync:2.1.6'
    implementation 'de.greenrobot:eventbus:2.4.0'
    implementation 'com.android.support:support-v13:25.0.3'
}

android {
    compileSdkVersion 23
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
```

```
    }

    aaptOptions {
        noCompress 'html'
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}
```

(from WebServer/Simple/app/build.gradle)

The Service

The bulk of the functionality lies in `WebServerService`, the awkwardly-named Android `Service` subclass that hosts the `AsyncHttpServer`.

The objective of `WebServerService` is to serve some Web content, specifically some content baked into the app via an `assets/` directory.

Setting Up the AsyncHttpServer

`onCreate()` on `WebServerService` does the basic plumbing of setting up the `AsyncHttpServer`:

```
  @Override
  public void onCreate() {
    super.onCreate();

    server=new AsyncHttpServer();
    server.get("/.*", new AssetRequestCallback());
    server.listen(4999);

    raiseStartedEvent();
    foregroundify();
  }
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

The `server` field holds our `AsyncHttpServer`, instantiated as part of `onCreate()`.

The `get()` call tells the `AsyncHttpServer` that we want to support HTTP `GET` requests on a particular URL regular expression. In this case, we use a wildcard to say that we are willing to entertain all URLs. `AssetRequestCallback` is an object that

will be given control when a matching GET request comes in and will handle sending back the response — we will examine this callback shortly. Note that AsyncHttpServer also has a post() convenience method (for HTTP POST requests) plus a more generic addAction() method for registering to support other sorts of HTTP operations (e.g., HEAD).

Once configured, we can call listen() on the AsyncHttpServer to set up the server to listen on the designated TCP/IP port (4999 in this case). There is also listenSecure() for supporting SSL, where you provide a SSLContext in addition to the port. Note that the server will be listening on all eligible network interfaces. For most Android devices, that will either be WiFi or mobile data.

The raiseStartedEvent() and foregroundify() calls will be explained in upcoming sections.

Serving Pages from Assets

AsyncHttpServer offers directory() methods that allow you to teach the server to automatically serve content from directories that you can access, such as some subdirectory of getFilesDir().

If you want to serve anything else, you will need to create an implementation of HttpServerRequestCallback and use that in your get(), post(), or addAction() calls. That callback object will be called with onRequest() whenever an HTTP request arrives that matches the HTTP verb and URI pattern you specified. You get an AsyncHttpServerRequest object that represents the request, and an AsyncHttpServerResponse that represents your response. Your job is to interpret the request and generate that response.

In the sample app, AssetRequestCallback is a HttpServerRequestCallback that looks in assets/, via AssetManager, for matching files and serves them:

```
private class AssetRequestCallback
  implements HttpServerRequestCallback {
  private final AssetManager assets;

  AssetRequestCallback() {
    assets=getAssets();
  }

  @Override
  public void onRequest(AsyncHttpServerRequest request,
```

```
                          AsyncHttpServerResponse response) {
    String path=request.getPath();

    try {
      if (path.length()==0 || "/".equals(path)) {
        path="index.html";
      }
      else if (path.startsWith("/")) {
        path=path.substring(1);
      }

      AssetFileDescriptor afd=getAssets().openFd(path);

      response.sendStream(afd.createInputStream(), afd.getLength());
    }
    catch (IOException e) {
      handle404(response, path);
    }
  }

  private void handle404(AsyncHttpServerResponse response,
                         String path) {
    Log.e(getClass().getSimpleName(),
      "Invalid URL: "+path);
    response.code(404);
    response.end();
  }
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

If your URI pattern contains wildcards — such as the `/.*` we used with the `get()` call in `onCreate()` — you can use `getPath()` on the `AsyncHttpServerRequest` to get the full path to the resource that the HTTP client is requesting. In this case, we normalize it a bit:

- If the path is empty or is purely `/`, we interpret this as trying to load the "home page" of the Web server and map that to an `index.html` file in `assets/`
- If the path begins with a `/`, we remove it, as `AssetManager` does not accept leading slashes when we later try to retrieve the asset using the `open()` method.

After that, we are in position to build our response. Ideally, for serving something out of assets, we would stream it right from storage, as opposed to reading the whole

thing into memory first. `AsyncHttpServerResponse` has a `sendStream()` method for just this purpose, taking an `InputStream` and the length of data to stream out.

It's that length that poses a problem. `AssetManager` has no direct way to get the length of an asset. So, while we could get our `InputStream` from the `AssetManager`, we still lack the length.

Instead, we use `openFd()` on `AssetManager` to open the asset as an `AssetFileDescriptor`. This has a `length()` method, to go along with the `createInputStream()` method. These we turn around and pass to `sendStream()`.

However, there is one big limitation here: the asset cannot be compressed. Otherwise, we will get an exception when trying to determine the length. By default, HTML files stored as assets will be compressed. So, back in our `app/` module's `build.gradle` file, we disable this:

```
aaptOptions {
  noCompress 'html'
}
```

This tells `aapt` (the tool responsible for putting stuff into the APK) not to compress files with a `.html` file extension.

Alternative options for sending results include:

- `send()`, which takes the MIME type and a `String` of the data for that MIME type
- `sendFile()`, which works great if the data to be returned is an existing file on the filesystem

`AsyncHttpServer` will attempt to guess a MIME type if you do not provide one, but it has a relatively limited list of known MIME types. You can use `setContentType()` to provide a MIME type separately, if you know it. Fortunately, in this case, it knows that `.html` files have a MIME type of `text/html`.

If we get an `IOException`, presumably the path that was requested does not match anything in assets, so we use the `code()` and `end()` methods on the `AsyncHttpServerResponse` to return an HTTP 404 (file not found) response.

Making a Foreground Service

Sometimes, you will only have your Web server running while your app is in the foreground. However, more often than not, you will want the Web server available longer than that. For example, when serving media to Chromecast for playback, you do not want to all of a sudden stop serving just because the user started doing something else on their device and your process was terminated while it was in the background.

Hence, there will be times when you will want your service to be a foreground service, so that Android is less likely to terminate it due to old age. The `foregroundify()` method that we called in `onCreate()` has a fairly basic recipe for setting up a foreground service:

```
private void foregroundify() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this);
  Intent iActivity=new Intent(this, MainActivity.class);
  PendingIntent piActivity=
    PendingIntent.getActivity(this, 0, iActivity, 0);
  Intent iReceiver=new Intent(this, StopReceiver.class);
  PendingIntent piReceiver=
    PendingIntent.getBroadcast(this, 0, iReceiver, 0);

  b.setAutoCancel(true)
    .setDefaults(Notification.DEFAULT_ALL)
    .setContentTitle(getString(R.string.app_name))
    .setContentIntent(piActivity)
    .setSmallIcon(R.mipmap.ic_launcher)
    .setTicker(getString(R.string.app_name))
    .addAction(R.drawable.ic_stop_white_24dp,
      getString(R.string.notify_stop),
      piReceiver);

  startForeground(1337, b.build());
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

Note that there are two `PendingIntent` objects associated with the `Notification`. If the user taps on the main portion of the notification tray tile, we will bring back the `MainActivity` to the foreground. However, we also add a "stop" action, tied to a `StopReceiver`. This manifest-registered `BroadcastReceiver` just calls `stopService()`, to shut down our service directly from the `Notification`:

```
package com.commonsware.android.webserver.simple;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class StopReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    context.stopService(new Intent(context, WebServerService.class));
  }
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/StopReceiver.java)

Raising Status Events

We want to let the UI layer know — if the UI layer exists at the moment – that the Web server has started and stopped. That way, the UI can adjust its presentation to reflect that fact, such as toggling action bar items between "play" and "stop" icons.

Also, when the Web server is started, we need to let the user know what URL(s) will work to communicate with that Web server. Devices do not usually get domain names, and it is aggravating for a user to find out the IP address(es) of the device. It would be simpler if our UI could inform the user of URLs that can reach the service, so that the user can more readily type those URLs in on a desktop Web browser (or whatever).

To that end, `WebServerService` has a pair of `static` classes that serve as events for EventBus: `ServerStartedEvent` and `ServerStoppedEvent`:

```
  static class ServerStartedEvent {
    private ArrayList<String> urls=new ArrayList<String>();

    void addUrl(String url) {
      urls.add(url);
    }

    ArrayList<String> getUrls() {
      return (urls);
    }
  }

  static class ServerStoppedEvent {
```

```
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

`ServerStartedEvent` not only is the indication that the Web server has started, but it also contains the URLs that the UI can display to the user.

The `raiseStartedEvent()` method called from `onCreate()` is responsible for raising the `ServerStartedEvent`:

```
private void raiseStartedEvent() {
  ServerStartedEvent event=new ServerStartedEvent();

  try {
    for (Enumeration<NetworkInterface> enInterfaces=
         NetworkInterface.getNetworkInterfaces();
         enInterfaces.hasMoreElements(); ) {
      NetworkInterface ni=enInterfaces.nextElement();

      for (Enumeration<InetAddress> enAddresses=
           ni.getInetAddresses();
           enAddresses.hasMoreElements(); ) {
        InetAddress addr=enAddresses.nextElement();

        if (addr instanceof Inet4Address) {
          event.addUrl(
            "http://"+addr.getHostAddress()+":4999");
        }
      }
    }
  }
  catch (SocketException e) {
    Log.e(getClass().getSimpleName(), "Exception in IP addresses", e);
  }

  EventBus.getDefault().removeAllStickyEvents();
  EventBus.getDefault().postSticky(event);
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

The bulk of this method is involved in determining the IP addresses for the device:

- Iterate over all of the network interfaces
- For each network interface, iterate over all of its IP addresses
- For each IP address, see if it is an IPv4 address

- For each IPv4 address, construct the proper URL (with scheme and port) and add it to a running list of URLs inside the `ServerStartedEvent`

In terms of the event itself, both `ServerStartedEvent` and `ServerStoppedEvent` will be sent using sticky events. These will allow our activity to be destroyed and recreated, yet still find out the status of the Web server. However, we only want *one* of these events to be outstanding — if the user starts, stops, then starts the Web server again, we do not want three sticky events floating around.

So, we remove all existing sticky events (if any), then use `postSticky()` to raise the `ServerStartedEvent`.

The Rest of the Lifecycle

`ServerStoppedEvent` is posted from `onDestroy()`:

```
@Override
public void onDestroy() {
  EventBus.getDefault().removeAllStickyEvents();
  EventBus.getDefault().postSticky(new ServerStoppedEvent());
  server.stop();
  AsyncServer.getDefault().stop(); // no, really, I mean stop

  super.onDestroy();
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

As was the case with `ServerStartedEvent`, we wipe out any existing sticky events (such as the `ServerStartedEvent` that should have been posted earlier), then post the `ServerStoppedEvent`.

To stop the Web server, we have to do two things:

1. Call `stop()` on the `AsyncHttpServer`, which ideally would be enough
2. Call `stop()` on the default `AsyncServer`, which is an unfortunately-required minor hassle

While we will use `startService()` to start the service, we are not using the command pattern to send commands to the service. In principle, that means we could skip the `onStartCommand()` method. However, the default implementation of `onStartCommand()` returns `START_STICKY`, which means that Android will keep trying to restart our service after our process gets terminated. This is a ridiculous

default value and goes a long way towards explaining Android's memory issues. So, we override `onStartCommand()` to return `START_NOT_STICKY`, to indicate that if the process is terminated, do *not* automatically restart the service:

```
@Override
public int onStartCommand(Intent i, int flags, int startId) {
  return(START_NOT_STICKY);
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

And, since this service does not support the binding pattern, we have a stub implementation of `onBind()` to satisfy the compiler, since `onBind()` is an `abstract` method on `Service`:

```
@Override
public IBinder onBind(Intent intent) {
  throw new UnsupportedOperationException("Go away");
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/WebServerService.java)

The Activity

The main activity — surprisingly enough, named `MainActivity` — is not that complicated, though it does have a couple of interesting wrinkles.

The activity itself is a `ListActivity`, where the URLs that are supplied in the `ServerStartedEvent` will be displayed as rows in the list. This particular app has no need for anything else in the content area, so we just inherit the full-screen `ListView` and skip `onCreate()` entirely.

As with many other samples using greenrobot's EventBus, `MainActivity` registers with the bus in `onResume()` and unregisters in `onPause()`:

```
@Override
protected void onResume() {
  super.onResume();

  EventBus.getDefault().registerSticky(this);
}

@Override
protected void onPause() {
  EventBus.getDefault().unregister(this);
```

```
    super.onPause();
  }
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

Note that since we are looking to use sticky events, we have to use `registerSticky()` rather than the ordinary `register()` method.

The user will be able to start and stop the Web server through action bar items, defined in `res/menu/actions.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/start"
    android:icon="@drawable/ic_phone_android_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_record"
    android:visible="true"/>
  <item
    android:id="@+id/stop"
    android:icon="@drawable/ic_stop_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_stop"
    android:visible="false"/>
</menu>
```

(from WebServer/Simple/app/src/main/res/menu/actions.xml)

The icons come from Google's material design icon roster.

The vision is that the `start` item will be visible when the Web server is not running, while the `stop` item will be visible when the Web server is running. That means we need to know whether the Web server is running when the activity is created, in addition to finding out changes in the Web server's state (e.g., user stopped it via the `Notification`). You might think that the Web server would not be running when the activity is created, but that ignores:

- Configuration changes (e.g., screen rotation)
- The user exiting the activity via BACK while the Web server is running, then returning to the activity via the launcher icon, overview screen (a.k.a., recent-tasks list), `Notification`, etc.

While our `ServerStartedEvent` is sticky, `onResume()` is called before `onCreateOptionsMenu()`. The `registerSticky()` call will immediately hand us the sticky event (if there is one). However, if we want to change the state of the two action bar items when the events arrive, we might not *have* those action bar items yet, if `onCreateOptionsMenu()` has not yet been called.

To handle all of this, we have the two `MenuItem` objects for those action bar items (`start` and `stop`) defined as fields on `MainActivity`, and they are populated in `onCreateOptionsMenu()` as normal:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  start=menu.findItem(R.id.start);
  stop=menu.findItem(R.id.stop);

  WebServerService.ServerStartedEvent event=
    EventBus.getDefault().getStickyEvent(WebServerService.ServerStartedEvent.class);

  if (event!=null) {
    handleStartEvent(event);
  }

  return(super.onCreateOptionsMenu(menu));
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

However, we *also* check to see if we have a sticky `ServerStartedEvent`; if yes, we call a private `handleStartEvent()` to toggle the state of the two action bar items plus load our URLs into the `ListView`:

```
private void handleStartEvent(WebServerService.ServerStartedEvent event) {
  start.setVisible(false);
  stop.setVisible(true);

  setListAdapter(new ArrayAdapter<String>(this,
    android.R.layout.simple_list_item_1, event.getUrls()));
}
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

We also call `handleStartEvent()` from `onEventMainThread()`, if and only if we have our action bar items set up (i.e., if `start` is not `null`):

```
public void onEventMainThread(WebServerService.ServerStartedEvent event) {
  if (start!=null) {
    handleStartEvent(event);
```

```
    }
  }
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

We also need to watch for `ServerStoppedEvent` events, so we can flip the action bar items and clear the list:

```
  public void onEventMainThread(WebServerService.ServerStoppedEvent event) {
    if (start!=null) {
      start.setVisible(true);
      stop.setVisible(false);
      setListAdapter(null);
    }
  }
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

However, since the stopped state is also the natural initial state of our activity, we do not need to do anything special to check for that event manually at startup.

If the user taps on either of those action bar items, we start or stop the service, in `onOptionsItemSelected()`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    Intent i=new Intent(this, WebServerService.class);

    if (item.getItemId()==R.id.start) {
      startService(i);
    }
    else {
      stopService(i);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

And, if the user happens to tap on one of those `ListView` rows containing a URL for the Web server, we will go start up a Web browser on the device itself to go view that URL, in `onListItemClick()`:

```
  @Override
  protected void onListItemClick(ListView l, View v, int position, long id) {
```

```
    startActivity(new Intent(Intent.ACTION_VIEW,
      Uri.parse(getListAdapter().getItem(position).toString())));
  }
```

(from WebServer/Simple/app/src/main/java/com/commonsware/android/webserver/simple/MainActivity.java)

The Results

Running the app brings up the initial UI, largely empty:

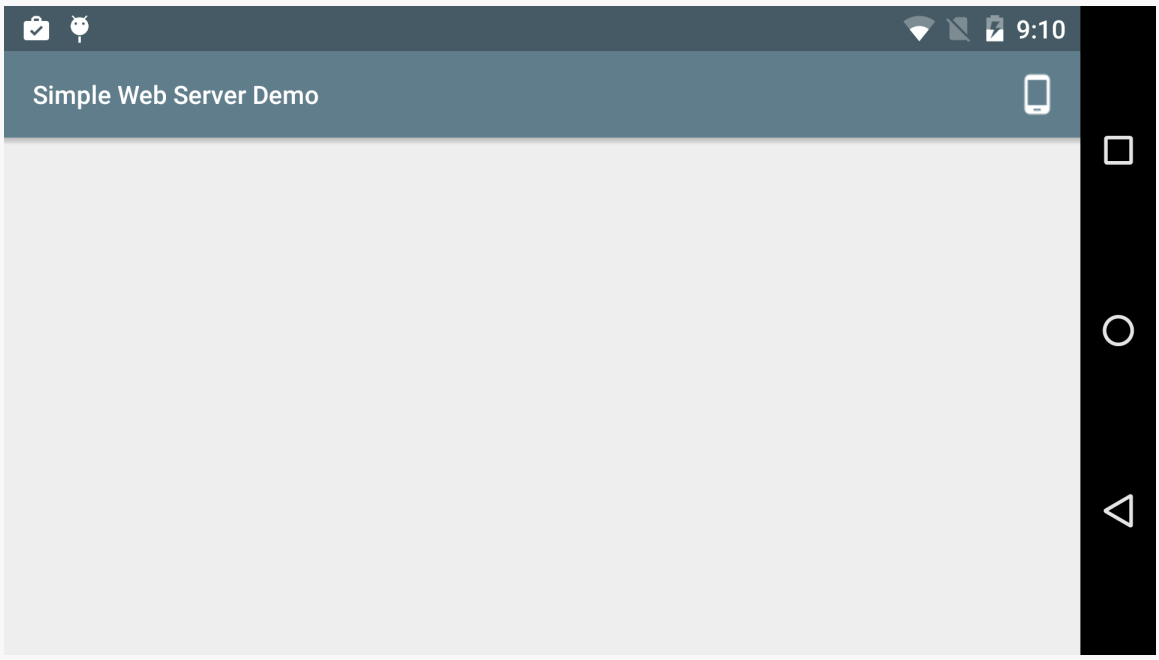


Figure 805: Simple Web Server Demo, As Initially Launched

Tapping the action bar item (icon looks like a phone) will start the Web server, and the UI will show the available URLs:

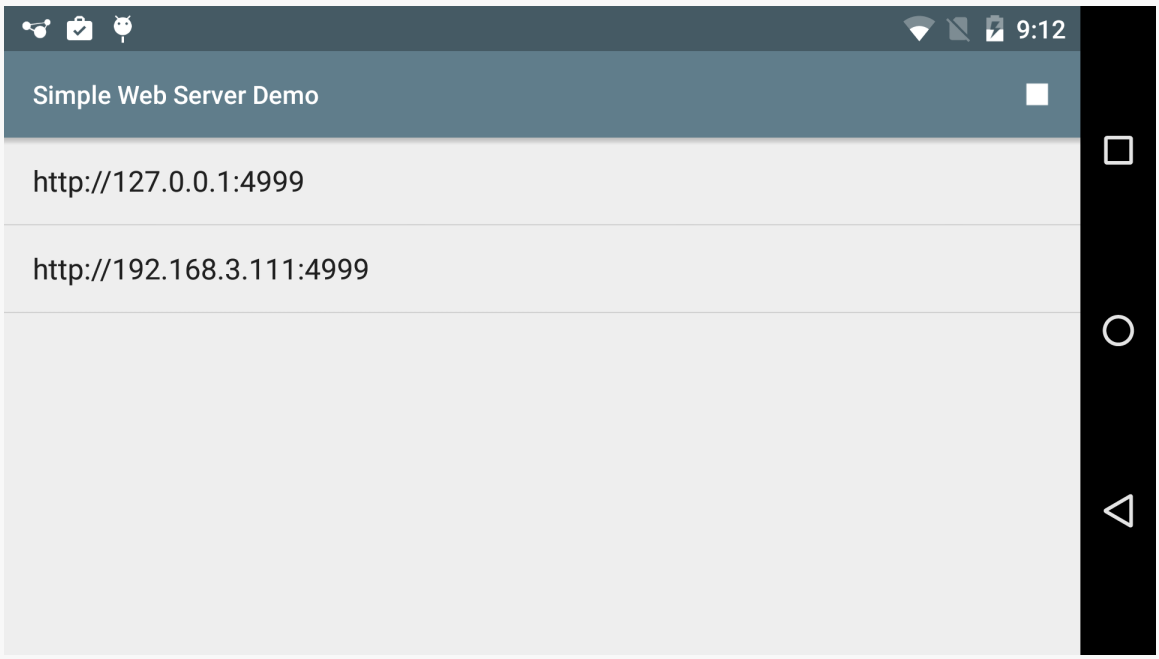


Figure 806: Simple Web Server Demo, With Web Server Running

If you are running this on hardware, you should be able to visit the Web server using the non-`localhost` URL (i.e., the one that is not for an IP address of `127.0.0.1`). Or, tap on any of the URLs in the UI to bring up a Web browser on the device or emulator for that URL:

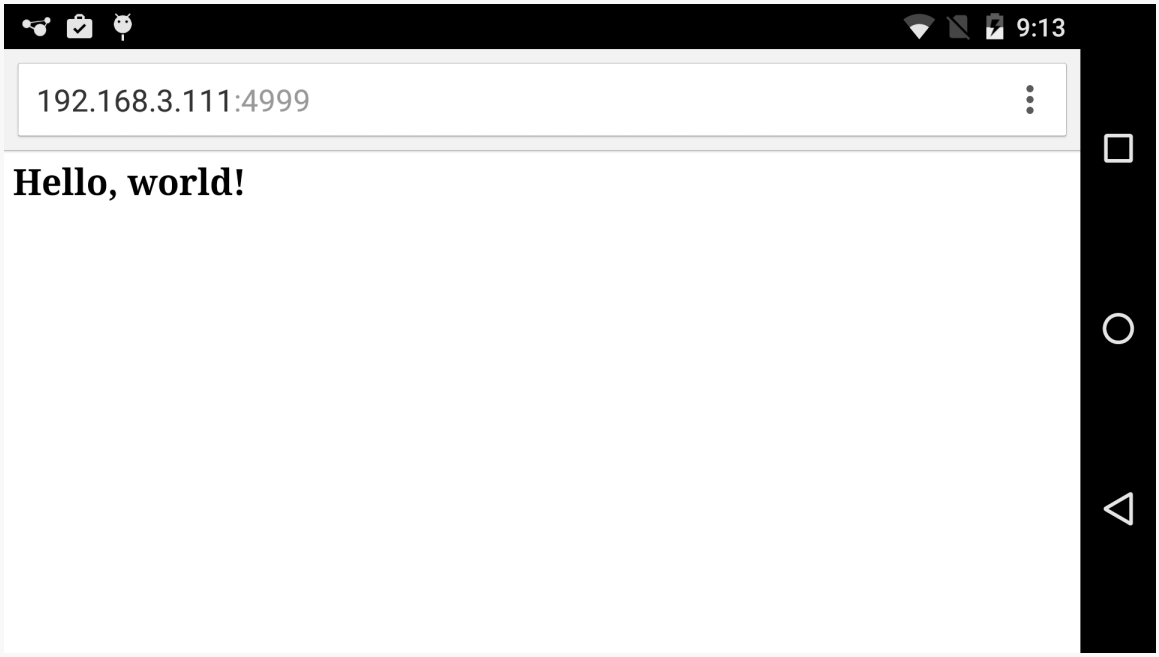


Figure 807: Web Page Served by Simple Web Server Demo

You can stop the server either by tapping the “stop” action bar item, or by clicking the “stop” action in the foreground `Notification`:

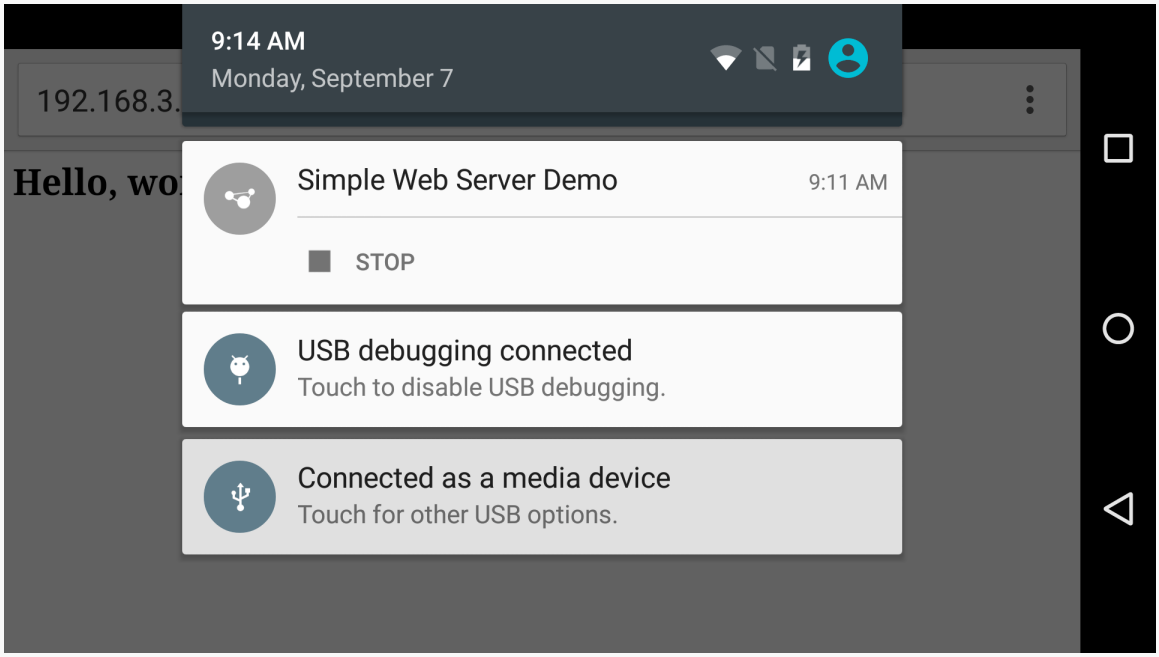


Figure 808: Simple Web Server Demo Notification, with Stop Action

Template-Driven Responses, with Handlebars

Many times, you will want to generate the HTML that you send back from the Web server to the browser, rather than use simple files or assets. From the standpoint of `AsyncHttpServer`, this is mostly a matter of having another `HttpServerRequestCallback` that handles the HTML generation process.

The sky is the (proverbial) limit in terms of how you generate that HTML, as you have the full power of Java at your disposal. However, in many cases, you will want to take advantage of template engines to simplify your work. Rather than concatenating together a seemingly infinite number of strings to assemble your HTML from parts, you have a template that pulls in dynamic bits as needed.

In the chapter on printing, you will see how to use the jmustache template engine for building HTML on the fly. The name "jmustache" is based on a popular template language syntax, using "mustaches" (braces) to represent the dynamic bits. So, `Hello, {{ username }}` would pull `username` from some supplied context and put it after `Hello,` in the resulting string.

Another engine that uses the same basic syntax, but with a bit more power behind it, is handlebars. The original handlebars implementation is in JavaScript, but there

is a Java port of it that we can use in our Android apps. In this section, we will look at how to add handlebars support to our miniature Web server, to return template-driven HTML content, as shown in the `WebServer/Template` sample application.

Adding the Handlebars Dependency

Edgar Espina's Handlebars.java library will be our template library. This library is available on Maven Central and so only requires that we add one line to `build.gradle` to pull in this dependency:

```
dependencies {
    implementation 'com.koushikdutta.async:androidasync:2.1.6'
    implementation 'de.greenrobot:eventbus:2.4.0'
    implementation 'com.android.support:support-v13:25.0.3'
    implementation 'com.github.jknack:handlebars:2.2.2'
}
```

(from WebServer/Template/app/build.gradle)

Loading Handlebars Templates

In `onCreate()` of the revised `WebServerService`, we need to initialize the template engine, by creating an instance of the `Handlebars` class, where we store that instance in a field of the service:

```
public class WebServerService extends Service {
  private AsyncHttpServer server;
  private Handlebars handlebars;
  private Template t;

  @Override
  public void onCreate() {
    super.onCreate();

    handlebars=new Handlebars(new AssetTemplateLoader(getAssets()));

    try {
      t=handlebars.compile("demo.hbs");
      server=new AsyncHttpServer();
      server.get("/demo", new TemplateRequestCallback());
      server.get("/.*", new AssetRequestCallback());
      server.listen(4999);

      raiseReadyEvent();
      foregroundify();
```

```
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
        "Exception starting Web server", e);
    }
  }
```

(from WebServer/Template/app/src/main/java/com/commonsware/android/webserver/template/WebServerService.java)

The parameter to the `Handlebars` constructor is a `TemplateLoader` that is responsible for loading templates from some data store. The library comes with loaders that can load from simple files. However, reminiscent of the previous sample, it would be nice if we could package our templates as assets, so that they ship with our app. Handlebars.java knows nothing about Android and assets, so we have to create our own `AssetTemplateLoader` that implements the `TemplateLoader` interface. We do this by extending the `AbstractTemplateLoader` base class and override `sourceAt()`. `sourceAt()` takes a `String` representation of the location of the template, and it is our job to return a `TemplateSource` that encapsulates the template itself. As there is no `AssetTemplateSource`, the simplest way to do this is to read the template into memory and wrap it in `StringTemplateSource`, which is what `AssetTemplateLoader` does:

```
  private static class AssetTemplateLoader
    extends AbstractTemplateLoader {
    private final AssetManager mgr;

    AssetTemplateLoader(AssetManager mgr) {
      this.mgr=mgr;
    }

    @Override
    public TemplateSource sourceAt(String s) throws IOException {
      return(new StringTemplateSource(s, slurp(mgr.open(s))));
    }
  }

  // inspired by http://stackoverflow.com/a/309718/115145

  public static String slurp(final InputStream is) throws IOException {
    final char[] buffer=new char[1024];
    final StringBuilder out=new StringBuilder();
    final InputStreamReader in=new InputStreamReader(is, "UTF-8");

    while (true) {
      int rsz=in.read(buffer, 0, buffer.length);
      if (rsz < 0)
```

```
        break;
      out.append(buffer, 0, rsz);
    }

    return(out.toString());
  }
```

(from WebServer/Template/app/src/main/java/com/commonsware/android/webserver/template/WebServerService.java)

By using `AssetTemplateLoader` when we create the `Handlebars` instance, we will be able to load templates out of the `main` source set's `assets/` folder, such as `demo.hbs`:

```
<html>
    <head><title>Display Metrics</title></head>
    <body>
        <h1>Display Metrics</h1>
        <table>
            <tr><th>Density DPI</th><td>{{ densityDpi }}</td></tr>
            <tr><th>Density xDPI</th><td>{{ xdpi }}</td></tr>
            <tr><th>Density yDPI</th><td>{{ ydpi }}</td></tr>
            <tr><th>Dimensions</th><td>{{ widthPixels }} x {{ heightPixels }}</td></tr>
        </table>
    </body>
</html>
```

(from WebServer/Template/app/src/main/assets/demo.hbs)

In that file, you will see a few table cells that will be filled in by dynamic data (e.g., `{{ densityDpi }}`). What this HTML page represents is a report of some of the key values from a `DisplayMetrics` object, providing details of resolution and density of the Android device's screen.

Back up in `onCreate()`, we go ahead and load this template, via a call to `compile()` on the `Handlebars` object. This `Template` instance is then ready to generate a custom page for us, once we give it the data to feed into that page.

Handlebars' Context (No, Not *That* Context)

The data for the handlebars-delimited macros in the template file (e.g., `{{ densityDpi }}`) comes from what Handlebars refers to as its context. To render a template, you create a context and apply it to the template.

Back up in `onCreate()`, when we set up our routes on our Web server, we added one for `/demo`, pointing to a `TemplateRequestCallback`:

```
      server.get("/demo", new TemplateRequestCallback());
```

(from WebServer/Template/app/src/main/java/com/commonsware/android/webserver/template/WebServerService.java)

This means that whenever the server gets a `/demo` request, `TemplateRequestCallback` is responsible for it, through the same sort of `onRequest()` callback method that we used in `AssetRequestCallback`:

```
private class TemplateRequestCallback implements HttpServerRequestCallback {
  @Override
  public void onRequest(AsyncHttpServerRequest request, AsyncHttpServerResponse response) {
    try {
      DisplayMetrics metrics=new DisplayMetrics();
      WindowManager wmgr=(WindowManager)getSystemService(WINDOW_SERVICE);

      wmgr.getDefaultDisplay().getMetrics(metrics);

      Context ctxt=Context
        .newBuilder(metrics)
        .resolver(FieldValueResolver.INSTANCE)
        .build();

      response.send(t.apply(ctxt));
      ctxt.destroy();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
        "Exception serving Web page", e);
    }
  }
}
```

(from WebServer/Template/app/src/main/java/com/commonsware/android/webserver/template/WebServerService.java)

Here, we first get our hands on a `DisplayMetrics` instance, as that is the source of the data that we want to pour into the response.

We then build a context, through a class unfortunately named `Context`. Since you cannot have two imports for `Context`, you will either be able to import `android.app.Context` *or* `com.github.jknack.handlebars.Context`, and you would have to refer to the other `Context` via the fully-qualified class name. In this case, `WebServerService` did not need `android.app.Context`, so a bare `Context` class name is referring to `com.github.jknack.handlebars.Context`.

There are many ways to populate a Handlebars `Context`. For the purposes of this example, we only need one source of data: the `DisplayMetrics` instance. `newBuilder()` is a factory method on `Context` that creates a `Context` builder object, providing some data source as a starting point. You can then further configure the builder, before eventually calling `build()` to get the `Context`. In this case, we call `resolver()` on the builder, to indicate how the contents of a macro translate into operations against our data source. Here, we are using `FieldValueResolver.INSTANCE`, which says that a macro like `{{ displayDpi }}` should be interpreted as a reference to a field on the data source instance.

We then use the `Context` to generate our result from the `Template` via the `apply()` method, and `send()` that result.

When you are done with a `Context`, call `destroy()` on it, per the Handlebars documentation.

The Results

If you run the app, start the server, and view the `/demo` page, you will see some values culled from `DisplayMetrics` on the device:

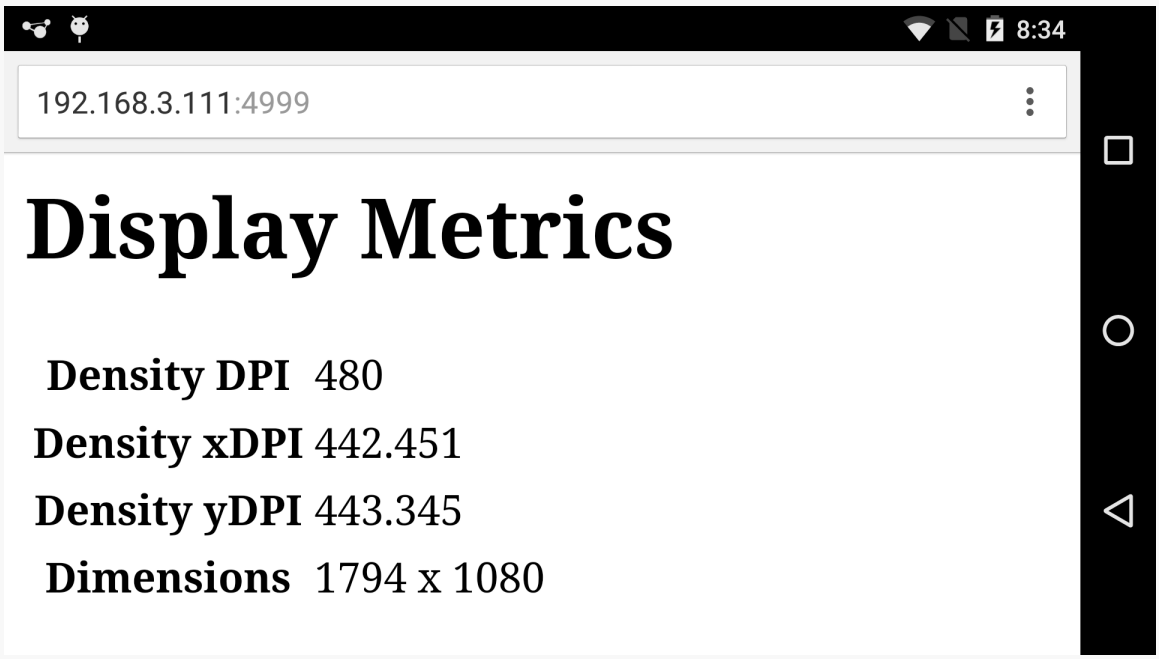


Figure 809: Template Web Server Demo, /demo URL Response, Zoomed

Supporting WebSockets

Given sufficient time and effort, there is nothing stopping you from building a full-fledged Web app served via an Android-hosted Web server.

One thing that many Web apps use today is a WebSocket. A WebSocket is a bi-directional data channel between client and server. In particular, it is used for "server push", where the server sends messages to the client, perhaps related to some data changes on the server.

AsyncHttpServer not only supports serving HTTP requests, but is also offers WebSocket support as well. The WebServer/WebSockets sample application builds upon the server created in the first sample app in this chapter, adding in a push channel so the server can asynchronously update the client.

Registering the WebSocket Listener

Just as AndroidAsync has HttpServerRequestCallback for handling HTTP requests, it also has AsyncHttpServer.WebSocketRequestCallback for handling incoming WebSocket connections. And, just as you register instances of HttpServerRequestCallback with an AsyncHttpServer to handle various HTTP request types (e.g., get(), post()), you register AsyncHttpServer.WebSocketRequestCallback instances with an AsyncHttpServer to handle those WebSocket connections.

This sample app's WebServerService has a slightly different onCreate() method than does the original. It calls websocket() on the AsyncHttpServer to tie a custom WebSocketClientCallback instance to the server, bound to a /ss URL. It also starts up a ScheduledExecutorService to get control every three seconds:

```
public class WebServerService extends Service implements Runnable {
  private AsyncHttpServer server;
  final private ArrayList<WebSocket> sockets=new ArrayList<WebSocket>();
  final private ScheduledExecutorService timer=
    Executors.newSingleThreadScheduledExecutor();

  @Override
  public void onCreate() {
    super.onCreate();

    server=new AsyncHttpServer();
    server.websocket("/ss", new WebSocketClientCallback());
    server.get("/.*", new AssetRequestCallback());
    server.listen(4999);

    raiseStartedEvent();
    foregroundify();

    timer.scheduleAtFixedRate(this, 3000, 3000, TimeUnit.MILLISECONDS);
  }
```

(from WebServer/WebSockets/app/src/main/java/com/commonsware/android/webserver/websockets/WebServerService.java)

We will use the `ScheduledExecutorService` as the trigger for sending messages over the WebSockets to any connected clients.

`WebSocketClientCallback` is an implementation of the `AsyncHttpServer.WebSocketRequestCallback` interface. It requires only one method: `onConnected()`, which is called when a new WebSocket connection has been requested by a client:

```
private class WebSocketClientCallback
  implements AsyncHttpServer.WebSocketRequestCallback {
  @Override
  public void onConnected(final WebSocket ws,
                          AsyncHttpServerRequest request) {
    sockets.add(ws);

    ws.setClosedCallback(new CompletedCallback() {
      @Override
      public void onCompleted(Exception ex) {
        if (ex!=null) {
          Log.e(getClass().getSimpleName(),
            "Exception with WebSocket", ex);
        }

        sockets.remove(ws);
      }
    });
  }
}
```

(from WebServer/WebSockets/app/src/main/java/com/commonsware/android/webserver/websockets/WebServerService.java)

Here, we do two things:

1. Add the `WebSocket` that we are passed to an `ArrayList` of outstanding `WebSocket` instances, being tracked in the service via a `sockets` field
2. Add a `CompletedCallback` to the `WebSocket`, to be notified if and when the client disconnects, so we can remove the `WebSocket` from the `sockets` `ArrayList` and, if there was some sort of exception that triggered the `WebSocket` to be closed, log that `Exception` to Logcat

Here, we are treating all outstanding `WebSocket` instances as equal. If you have the notion of different clients needing different messages pushed to them, you would need to hold onto the `WebSocket` instances in some other data structure (e.g., a

HashMap, keyed by something, to allow you to find the WebSocket associated with a given client).

Posting Messages to Clients

Sending a message on a WebSocket is trivial: call send() on the WebSocket instance, passing a string representing the message to be sent.

In onCreate(), we called scheduleAtFixedRate() to arrange to get control every three seconds. Our WebServerService implements the Runnable interface, so it can get control directly on those three-second intervals, in a run() method:

```
  @Override
  public void run() {
    for (WebSocket socket : sockets) {
      socket.send(new Date().toString());
    }
  }
```

(from WebServer/WebSockets/app/src/main/java/com/commonsware/android/webserver/websockets/WebServerService.java)

Once again, we are treating all of the WebSocket instances the same, sending the same message to each: the current date and time. Obviously, a more sophisticated app might do something more elaborate here, such as sending over a JSON-formatted string containing a complex data structure.

Receiving Messages on the Client

In this sample app, we want the client to be a Web browser, one that is viewing our index.html page, courtesy of the normal Web serving feature of AsyncHttpServer. That index.html is a bit different than the one we used in the first sample:

```
<html>
<head>
    <title>CommonsWare Android WebSocket Server Demo</title>
</head>
<body>
<h1>Messages from Server</h1>
<ul id="transcript"></ul>
<script src="app.js"></script>
</body>
</html>
```

(from WebServer/WebSockets/app/src/main/assets/index.html)

Of note:

- We have an empty bulleted list, with an `id` of `transcript`
- We load an `app.js` file from the same server, where that file contains our WebSocket client code:

```
window.onload = function() {
    var ws_url=location.href.replace('http://', 'ws://')+'ss';
    var socket=new WebSocket(ws_url);

    socket.onopen = function(event) {
      // console.log(event.currentTarget.url);
    };

    socket.onerror = function(error) {
      console.log('WebSocket error: ' + error);
    };

    socket.onmessage = function(event) {
      var li=document.createElement("li");

      li.appendChild(document.createTextNode(event.data));
      document.getElementById("transcript").appendChild(li);
    };
}
```

(from WebServer/WebSockets/app/src/main/assets/app.js)

This book is not here to provide an extensive description of JavaScript-based WebSocket client development. What this snippet of JavaScript does is arrange to get control when the page is loaded (`window.onload`). At that point, it derives the URL for the WebSocket endpoint on the server, by changing the scheme (from `http` to `ws`, the official IETF scheme for WebSockets) and appending the `ss` to the end of the URL.

NOTE: this code is very simplistic and assumes that the URL used to load this Web page is simply the home page (e.g., `http://AAA.BBB.CCC.DDD/`, where AAA.BBB.CCC.DDD is the Android device's IP address). A more robust implementation would do a better job.

After creating a `WebSocket` object for that URL, we register three event handlers:

- `onopen`, which is called when we have opened the WebSocket successfully
- `onerror`, which is called if there is some sort of problem with the WebSocket

- `onmessage`, which is called when we receive a message from the server

In `onmessage`, we create a `<li>` element, slap the message from the server into the text of that element, and append it to the `transcript`.

The result, if you load the home page in a reasonably modern browser, is the timestamps of when the server got control via the `ScheduledExecutorService`, showing up in a bulleted list, in chronological order:

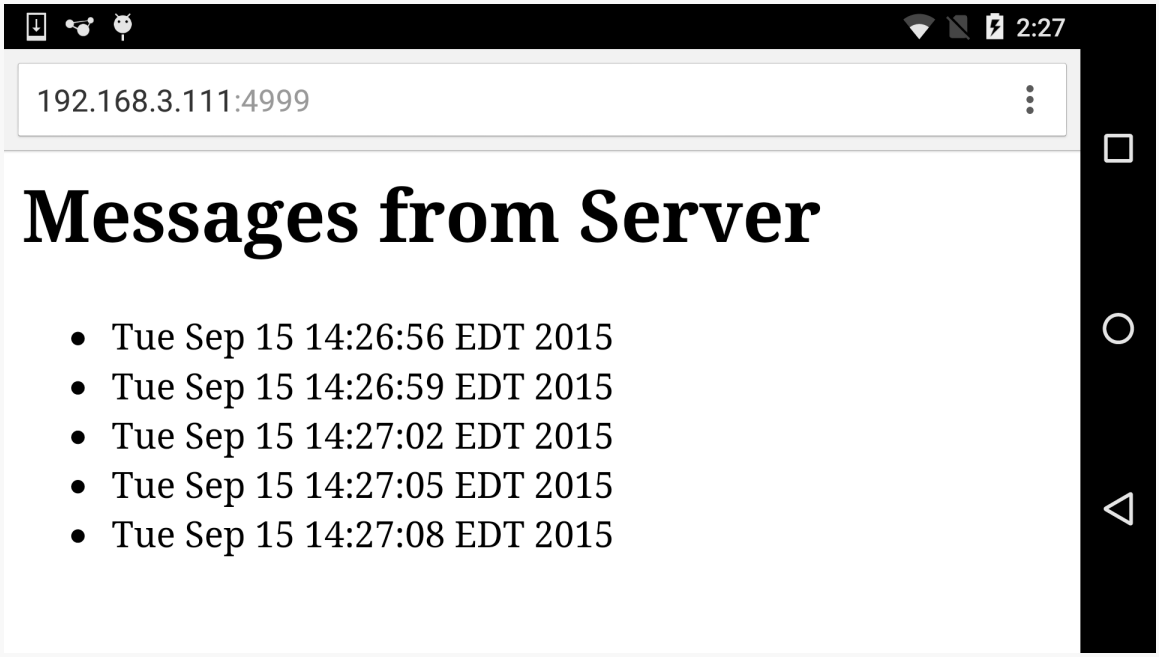


Figure 810: WebSocket Server Demo, Zoomed

Reversing the Communications Flow

Of course, nothing is stopping you from having data flow in the other direction, from the client to the server.

In JavaScript, there is a `send()` method on the `WebSocket` object that you can use to send a `String` to the server.

On the server side, you can call `setStringCallback()` to register a `StringCallback` implementation, which will be called with `onStringAvailable()` whenever a string message arrives from that WebSocket's client.

Implementing a WebSocket Client in Android

While this sample was focused on a browser as a client, AndroidAsync also has a WebSocket client API. Some sort of same-LAN peer-to-peer Android app might use this with another Android app's WebSocket server. Or, you could communicate with arbitrary WebSocket servers out on the Internet.

The AndroidAsync documentation has more on this process.

Securing the Web Server

However, as has been emphasized throughout this chapter, security is an issue any time you have open ports on a mobile device. Even if you think that your use of this Web server will be only for debugging purposes, developers are people too, and people make mistakes.

We can do a bit more to "harden" the Web server, to make it a bit more robust in the face of threats and user error. The `WebServer/Secure` sample application — starting from the `WebServer/Simple` project – demonstrates a few of the techniques.

Disabling on Mobile Data Connections

There are several types of network connection supported by Android. The two that get the most attention are WiFi and mobile data, but some devices offer others (e.g., wired Ethernet).

The one connection type that is the riskiest from a security standpoint is the mobile data connection. With WiFi, usually you will be behind some firewall, or at the very least a NAT-equipped router, which will limit the scope of attacks to anyone that can get to that WiFi LAN segment. Usually, that will only be people on that same WiFi network, which limits the scope of who could attack you.

With mobile data, though, at best you are part of a network with arbitrary other people on it (other mobile subscribers), and at worst you are given a *public* IP address and anyone in the world can reach your server.

Besides, usually mobile data is not a useful connection type for a mobile Web server. If the IP address given to the device by the mobile carrier is a private IP address, most browsers cannot get to that device, such as the user's own desktop browser.

Hence, it seems reasonable to block attempts to start the server when we are on a mobile network. It also seems reasonable to block attempts to start the server if we have no network connection at all, as the server may not be that useful in this state.

So, `onCreate()` of `WebServerService` checks to see what we are on and reacts accordingly:

```
  ConnectivityManager mgr=(ConnectivityManager)getSystemService(CONNECTIVITY_SERVICE);
  NetworkInfo ni=mgr.getActiveNetworkInfo();

  if (ni==null || ni.getType()==ConnectivityManager.TYPE_MOBILE) {
    EventBus.getDefault().post(new ServerStartRejectedEvent());
    stopSelf();
  }
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

We use `ConnectivityManager` to check for the active network (`getActiveNetworkInfo()`). If we have no connection (`getActiveNetworkInfo()` returns `null`) or it is a mobile data connection (`getType()` returns `TYPE_MOBILE`), we raise a `ServerStartRejectedEvent` and stop the service. Our activity can register for a `ServerStartRejectedEvent` and do something to let the user know that the Web server is not running.

Note that there is an `else` following the `if` from the above code – we will examine the full `onCreate()` method a bit later in this chapter.

Implementing an Inactivity Timeout

What happens if the user fails to stop the Web server? The longer the server is running, the more likely it is that somebody will discover and attempt to attack it.

However, we can implement an inactivity timeout. If we do not receive a valid HTTP request within X period of time, we automatically stop the service.

`WebServerService` defines a `MAX_IDLE_TIME_SECONDS` constant for how long we can go without a valid request before we stop the service:

```
  private static final int MAX_IDLE_TIME_SECONDS=60;
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

`WebServerService` also has a Java `ScheduledExecutorService`, which will keep track of when the timeout period is reached:

```
private ScheduledExecutorService timer=
  Executors.newSingleThreadScheduledExecutor();
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

In `onCreate()` — if we did not stop the service due to being on the wrong network — we arrange to get control at the designated time using the `ScheduledExecutorService`:

```
    timeoutFuture=timer.schedule(onTimeout,
      MAX_IDLE_TIME_SECONDS, TimeUnit.SECONDS);
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

We hold on to the response from `schedule()` — a `ScheduledFuture` object — in a field for later use. The `onTimeout` parameter is a simple `Runnable` that will be invoked `MAX_IDLE_TIME_SECONDS` from when `schedule()` was called:

```
private Runnable onTimeout=new Runnable() {
  @Override
  public void run() {
    stopSelf();
  }
};
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

The net effect is that `MAX_IDLE_TIME_SECONDS` from when the service is created, we stop it.

However, so long as the Web server is active, we do *not* want to stop it. To handle that, `onRequest()` of `AssetRequestCallback` reschedules the timeout, if we have a successful request (e.g., not a 404):

```
      timeoutFuture.cancel(false);
      timeoutFuture=timer.schedule(onTimeout,
        MAX_IDLE_TIME_SECONDS, TimeUnit.SECONDS);
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

So, the service will stop `MAX_IDLE_TIME_SECONDS` from either when the service starts or from our last valid HTTP request, whichever comes last.

If you run the sample, start the server, and let it sit for a while, you will see that the server automatically stops. Since our logic for updating the UI is triggered by `onDestroy()`, we do not need to do anything special for this timeout shutdown.

Supporting Random URLs

Another thing we can do is change up our URLs. Rather than using simple paths like `/` or `/index.html`, we can add a dynamically-generated random prefix, like `/AG78`. This will make it more difficult for an attacker to get a valid page in response to some HTTP request, as they will have to guess the prefix in addition to the rest of the path. It makes things incrementally harder for the user, as they will have to enter in a few additional characters for the URL, but since navigating the content should be through hyperlinks and the like once the initial URL is used, the cost should not be too bad.

To do this, the revised `WebServerService` employs `SecureRandom`, a class that ties into high-quality (on newer versions of Android) random number generation algorithms. We also track the prefix in a `String` named `rootPath`:

```
  private SecureRandom rng=new SecureRandom();
  private String rootPath;
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

The `onCreate()` method then uses the `SecureRandom` object to generate a 20-bit `BigInteger` and converts that into a base-24 string. 20 bits means about a million possible prefix values, which should be enough. Using base-24 instead of base-10 reduces the number of characters that the user has to type, while avoiding any potential O (capital O) versus 0 (zero) confusion, as the letters used will be in the range from A through N.

```
@Override
public void onCreate() {
  super.onCreate();

  ConnectivityManager mgr=(ConnectivityManager)getSystemService(CONNECTIVITY_SERVICE);
  NetworkInfo ni=mgr.getActiveNetworkInfo();

  if (ni==null || ni.getType()==ConnectivityManager.TYPE_MOBILE) {
    EventBus.getDefault().post(new ServerStartRejectedEvent());
    stopSelf();
  }
  else {
    rootPath=
      "/"+new BigInteger(20, rng).toString(24).toUpperCase();

    server=new AsyncHttpServer();
```

```
    server.get("/.*", new AssetRequestCallback());
    server.listen(4999);

    raiseReadyEvent();
    foregroundify();
    timeoutFuture=timer.schedule(onTimeout,
      MAX_IDLE_TIME_SECONDS, TimeUnit.SECONDS);
  }
}
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

`onRequest()` in the `AssetRequestCallback` needs to remove this `rootPath` if the requested URL begins with it, so we do not try using it as part of looking up the associated asset. Conversely, if the requested URL does *not* begin with `rootPath`, it is an invalid URL, so we can return a 404 response to the request.

```
  @Override
  public void onRequest(AsyncHttpServerRequest request,
                        AsyncHttpServerResponse response) {
    String path=request.getPath();

    try {
      if (path.startsWith(rootPath)) {
        path=path.substring(rootPath.length()+1);
      }
      else {
        handle404(response, path, null);
        return;
      }

      if (path.length()==0 || "/".equals(path)) {
        path="index.html";
      }
      else if (path.startsWith("/")) {
        path=path.substring(1);
      }

      AssetFileDescriptor afd=getAssets().openFd(path);

      response.sendStream(afd.createInputStream(),
        afd.getLength());
      timeoutFuture.cancel(false);
      timeoutFuture=timer.schedule(onTimeout,
        MAX_IDLE_TIME_SECONDS, TimeUnit.SECONDS);
      invalidRequestCount=0;
    }
    catch (IOException e) {
      handle404(response, path, e);
```

```
    }
  }
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

The `invalidRequestCount` referred to late in the `onRequest()` implementation is part of some code for detecting attackers, which we get into in the next section.

Detecting Attacks

Having a random prefix makes it more difficult for an attacker to get a valid URL. However, if they can just keep trying, eventually they will hit upon the same prefix that we are using and will be able to get Web pages from the server.

The obvious defense would be to block requests from clients that seem to be attempting to guess URLs, by counting the number of invalid requests that they make and rejecting future requests outright once they exceed some threshold.

In the sample app, we take an even more draconian approach: if somebody is attacking us, stop the service entirely. This has the side effect of stopping legitimate requests as well, of course.

In the sample app, there are two types of invalid request:

1. Requests with the proper prefix but asking for a path within there that does not match any of our assets
2. Requests without the proper prefix

Given that our `AssetRequestCallback` is handling the wildcard path (`/.*`), both types of invalid request will come to `AssetRequestCallback`. And, in both cases, `handle404()` is called.

So, `WebServerService` has a field named `invalidRequestCount`, to track how many sequential invalid requests are made. In `handle404()`, we increment that count by calling a `trackInvalidRequests()` method back up on `WebServerService`:

```
  private void handle404(AsyncHttpServerResponse response,
                         String path, Exception e) {
    Log.e(getClass().getSimpleName(), "Invalid URL: "+path, e);
    response.code(404);
    response.end();
    trackInvalidRequests();
  }
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

`trackInvalidRequests()` increments the count and, if the value exceeds a certain threshold, stops the service:

```
private void trackInvalidRequests() {
  invalidRequestCount++;

  if (invalidRequestCount>MAX_SEQUENTIAL_INVALID_REQUESTS) {
    stopSelf();
  }
}
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

The `invalidRequestCount=0` line at the bottom of `onRequest()` resets this counter, as we are tracking *sequential* invalid requests. This means that a user who fumbles around a bit trying to enter the URL is not harmed long-term, once the correct URL is used. However, since most attackers tend to make attempts in rapid-fire fashion, several consecutive failures will trip the detection algorithm and shut down the server.

Here, the threshold is `MAX_SEQUENTIAL_INVALID_REQUESTS`, defined as 10:

```
private static final int MAX_SEQUENTIAL_INVALID_REQUESTS=10;
```

(from WebServer/Secure/app/src/main/java/com/commonsware/android/webserver/secure/WebServerService.java)

One key limitation with this approach is that it requires that all URLs be handled by some code of ours. There is no obvious way with `AsyncHttpServer` to find out if URLs we elect to have the server handle itself fail with a 404 or other error. With luck, this will be added in the future.

What About SSL?

In principle, you could have the embedded Web server use SSL for encrypting its traffic. `AsyncHttpServer` has a `listenSecure()` method that takes an `SSLContext` as a parameter, where you would configure the SSL certificates to use for your server.

However, in practice, SSL is not going to be that useful, except in select scenarios. Android devices do not generally have domain names, and traditional SSL certificates are tied to a domain name. While you could create a self-signed certificate, Web browsers will raise all sorts of warnings when users try visiting that site using a regular Web browser, harming usability.

The primary situation where using a self-signed certificate can work well is when non-browser code is serving as the client, particularly if that client code is your own app running on another device. Your app, serving in the role of the client, can validate that the served-up self-signed certificate is indeed the proper one, rather than simply failing or otherwise rejecting the self-signed certificate.

Towards a Reusable Web Server Service

A few samples in this book use an embedded Web server for a separate input and control surface, showing output or collecting input from a developer in a browser so as not to interfere with what is going on with the device itself. Rather than have each of those samples roll its own embedded Web server, we can combine the elements of the samples from this chapter into a reusable library module that the other projects can pull in and extend. The `WebServer/Reusable` project contains a `webserver` module that serves in this role.

Gradle Changes

Previous editions of this code were in application projects. Now, we need a library project.

Hence, the module's `build.gradle` file not only contains references to all of the dependencies needed from all of the preceding samples, but it also uses the `com.android.library` Android Gradle Plugin:

```
apply plugin: 'com.android.library'

dependencies {
    compile 'com.koushikdutta.async:androidasync:2.1.6'
    compile 'de.greenrobot:eventbus:2.4.0'
    compile 'com.android.support:support-v13:26.1.0'
    compile 'com.github.jknack:handlebars:2.2.2'
}

android {
    compileSdkVersion 26

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 26
    }
}
```

(from WebServer/Reusable/webserver/build.gradle)

Configuration via Abstract Methods

Hard-coded constants are not a great solution for reusable code, as those constants cannot be changed by somebody reusing the code. There are any number of patterns that can be used for configuration (e.g., `Builder` object populating some configuration data structure).

In this case, since we are providing a base class for others to override, we can use the simple approach of `abstract` methods.

So, in this version of the sample, `WebServerService` is `abstract` and defines five `abstract` methods:

```
abstract public class WebServerService extends Service {
  abstract protected void buildForegroundNotification(NotificationCompat.Builder b);
  abstract protected boolean configureRoutes(AsyncHttpServer server);
  abstract protected int getPort();
  abstract protected int getMaxIdleTimeSeconds();
  abstract protected int getMaxSequentialInvalidRequests();
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

buildForegroundNotification()

It used to be that `foregroundify()` configured the entire `Notification` used for the foreground service. Now, some of that stuff should be configured by the subclass, such as text and icons.

The revised `foregroundify()` will call out to `buildForegroundNotification()`, where the subclass can add in basic stuff to the `NotificationCompat.Builder`:

```
  private void foregroundify() {
    NotificationCompat.Builder b=
      new NotificationCompat.Builder(this);

    Intent iReceiver=new Intent(this, StopReceiver.class);
    PendingIntent piReceiver=
      PendingIntent.getBroadcast(this, 0, iReceiver, 0);

    b.setAutoCancel(true)
      .setDefaults(Notification.DEFAULT_ALL);

    buildForegroundNotification(b);
```

```
    b.addAction(R.drawable.ic_stop_white_24dp,
      getString(R.string.notify_stop),
      piReceiver);

    startForeground(NOTIFY_ID, b.build());
  }
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

Things configured on the `Builder` before calling `buildForegroundNotification()` could be overridden by the subclass. Things configured on the `Builder` *after* calling `buildForegroundNotification()`, on the other hand, are enforced by the base `WebServerService` class. In particular, we ensure that the "stop" action is there, pointing at our `StopReceiver`.

configureRoutes()

`onCreate()` used to handle everything with respect to configuring the routes of the Web server (i.e., what paths route to what handlers). Now, that is mostly to be handled by subclasses, via a `configureRoutes()` method:

```
@Override
public void onCreate() {
  super.onCreate();

  ConnectivityManager mgr=(ConnectivityManager)getSystemService(CONNECTIVITY_SERVICE);
  NetworkInfo ni=mgr.getActiveNetworkInfo();

  if (ni==null || ni.getType()==ConnectivityManager.TYPE_MOBILE) {
    EventBus.getDefault().post(new ServerStartRejectedEvent());
    stopSelf();
  }
  else {
    handlebars=new Handlebars(new AssetTemplateLoader(getAssets()));
    rootPath=
      "/"+new BigInteger(20, rng).toString(24).toUpperCase();

    server=new AsyncHttpServer();

    if (configureRoutes(server)) {
      server.get("/.*", new AssetRequestCallback());
    }

    server.listen(getPort());

    raiseReadyEvent();
    foregroundify();
    timeoutFuture=timer.schedule(onTimeout,
      getMaxIdleTimeSeconds(), TimeUnit.SECONDS);
  }
}
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

If `configureRoutes()` returns `true`, that means that `WebServerService` should add the all-wildcard route, pulling matching content from `assets/`, and tracking failures for the consecutive-invalid-request defense.

getPort() and getMaxIdleTimeSeconds()

The `onCreate()` code shown above also shows two more `abstract` methods that subclasses provide:

- `getPort()`, to return the port to use to run the Web server
- `getMaxIdleTimeSeconds()`, to return how long the server can be idle before it is automatically shut down

These replace constants found in earlier versions of the sample.

getMaxSequentialInvalidRequests()

Similarly, the subclass needs to supply a `getMaxSequentialInvalidRequests()` implementation, to return how many consecutive invalid requests are allowed before the Web server shuts down as a defensive measure. This is used by the slightly-revised `trackInvalidRequests()` method:

```
protected void trackInvalidRequests() {
  invalidRequestCount++;

  if (invalidRequestCount>getMaxSequentialInvalidRequests()) {
    stopSelf();
  }
}
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

Integrating WebSocket and Handlebars

The support for WebSockets was pulled over into the `Reusable` project largely as-is from its original implementation. The idea is that the `WebServerService` would handle basic registration of clients, leaving the subclass to push messages to them. To that end, the `sockets` collection of outstanding `WebSocket` instances is exposed to subclasses via a `protected` `getWebSockets()` method.

Also, since not all uses of `WebServerService` need WebSocket support, subclasses need to call a `serveWebSockets()` method to set up the WebSocket route. This method takes two parameters:

- the relative path, under the randomly-generated root path, to use for WebSocket registration, and
- a `WebSocketRequestCallback` instance for handling those registrations, or `null` to use the standard implementation supplied by `WebServerService`

```
protected void serveWebSockets(String relpath,
                               AsyncHttpServer.WebSocketRequestCallback cb) {
  StringBuilder route=new StringBuilder(rootPath);

  if (!relpath.startsWith("/")) {
    route.append('/');
  }

  route.append(relpath);

  if (cb==null) {
    cb=new WebSocketClientCallback();
  }

  server.websocket(route.toString(), cb);
}
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

`WebServerService` also integrates Handlebars support, creating the `Handlebars` instance in `onCreate()`, pulling the `Handlebars` templates from assets as before.

However, rather than having a dedicated route for Handlebars templates, the stock `AssetRequestCallback` now knows that paths ending in `.hbs` represent Handlebars templates. `AssetRequestCallback` will call a `getContextForPath()` method, supplying the path to the template (minus the randomly-generated root path), so subclasses can prepare and return the appropriate `Context` for resolving any macros encoded in the template:

```
private class AssetRequestCallback
  implements HttpServerRequestCallback {
  private final AssetManager assets;

  AssetRequestCallback() {
    assets=getAssets();
  }
```

```
@Override
public void onRequest(AsyncHttpServerRequest request,
                      AsyncHttpServerResponse response) {
  String path=request.getPath();

  try {
    if (path.startsWith(rootPath)) {
      path=path.substring(rootPath.length()+1);
    }
    else {
      handle404(response, path, null);
      return;
    }

    if (path.length()==0 || "/".equals(path)) {
      path="index.html";
    }
    else if (path.startsWith("/")) {
      path=path.substring(1);
    }

    if (path.endsWith(".hbs")) {
      Template t=handlebars.compile(path);
      Context ctxt=getContextForPath(path);

      response.send(t.apply(ctxt));
      response.setContentType("text/html");
      ctxt.destroy();
    }
    else {
      AssetFileDescriptor afd=assets.openFd(path);

      response.sendStream(afd.createInputStream(),
        afd.getLength());
    }

    resetTimeout();
    invalidRequestCount=0;
  }
  catch (IOException e) {
    handle404(response, path, e);
  }
}
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

However, `getContextForPath()` is not `abstract`, so subclasses that do not need Handlebars support do not need to worry about it. Instead, the stock implementation just throws an `IllegalStateException`, to make it obvious to developers that they need to override this method if they have `.hbs` files that are being served:

```
protected Context getContextForPath(String relpath) {
  throw new IllegalStateException("You need to override this if using Handlebars!");
}
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/WebServerService.java)

Stopping the Service

Mostly, stopping the subclass of `WebServerService` is not a problem. The service can call `stopSelf()`, or activities of apps that reuse `WebServerService` can call `stopService()` with an `Intent` identifying the subclass.

However, there is one problem area: `StopReceiver`. This `BroadcastReceiver` is used to handle the "stop" action added to the foreground `Notification`. It cannot call `stopSelf()`, as it is not the service. However, the original implementation of `StopReceiver` also does not work, as it calls `stopService()` on an `Intent` identifying `WebServerService`, and that is not the running service — some subclass of `WebServerService` is.

There are any number of possible solutions to this problem:

- We could skip the "stop" action altogether and have apps using this library deal with it. However, some developers might skip having a "stop" action, and that action is important for users.
- We could have another `abstract` method, where subclasses have to provide an `Intent`, or possibly just a Java `Class`, identifying the service.
- We could use Java reflection to try to find subclasses of `WebServerService` in our VM and stop those. This would require the least work on behalf of users of the library. However, this is an Android book, and so it would be nice to find a more "Android-y" solution.

This sample takes another approach: manual `Intent` resolution.

Apps using the library should have their `<service>` have an `<intent-filter>` in the manifest with an `<action>` of `com.commonsware.android.webserver.WEB_SERVER_SERVICE`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  package="com.commonsware.andprojector"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

  <application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".MainActivity"
      android:theme="@style/AppTheme">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <service
      android:name=".ProjectorService"
      android:exported="false">
      <intent-filter>
        <action android:name="com.commonsware.android.webserver.WEB_SERVER_SERVICE"/>
      </intent-filter>
    </service>
  </application>

</manifest>
```

(from MediaProjection/andprojector/app/src/main/AndroidManifest.xml)

`StopReceiver` then uses `PackageManager` to find all services in this package that implement that action string, creates `Intent` objects identifying them, and stops those services:

```
package com.commonsware.android.webserver;

import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;

public class StopReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Intent i=
      new Intent(context.getString(R.string.service_action))
        .setPackage(context.getPackageName());
```

```
    PackageManager mgr=context.getPackageManager();

    for (ResolveInfo ri : mgr.queryIntentServices(i, 0)) {
      ComponentName cn=
        new ComponentName(ri.serviceInfo.applicationInfo.packageName,
          ri.serviceInfo.name);
      Intent stop=new Intent().setComponent(cn);

      context.stopService(stop);
    }
  }
}
```

(from WebServer/Reusable/webserver/src/main/java/com/commonsware/android/webserver/StopReceiver.java)

You might wonder why the `<action>` element does not refer to the same string resource that `StopReceiver` uses when creating the `Intent`. Ideally, it would. However, this is not supported — action strings must be literal strings, not references to string resources.

Trimming Back the Project

Since `WebServerService` is `abstract`, we do not need it in our manifest. And, since the overall project is now a library project, we can trim the manifest back a fair bit:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  package="com.commonsware.android.webserver"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

  <application>
    <receiver android:name=".StopReceiver"/>
  </application>

</manifest>
```

(from WebServer/Reusable/webserver/src/main/AndroidManifest.xml)

Similarly, the library no longer has its activity (to be supplied by the apps using the library) or any of its resources.

Reusing the Module via Relative Paths

In theory, this library project could be published as an AAR in an artifact repository. That could be a local repo, or a remote one; the remote one could be public (e.g., Maven Central) or private to an organization.

A lightweight, though somewhat clunky, alternative is to have an app wishing to use the library have what amounts to a "virtual module" for the library.

In an project's `settings.gradle`, you normally just list the modules in the project itself. However, it is also possible to list modules that live somewhere else, updating the default location for that module to point to the proper spot:

```
include ':app', ':webserver'

project(':webserver').projectDir=new File('../../WebServer/Reusable/webserver')
```

Here, we declare that the project not only has an `app` module in its default location (`app/`), but that we have a `webserver` module whose files are in a location relative to the current project. Note that the path is relative to the project root and must point to the module we wish to reference (`webserver/`), not just the project containing that module.

Then, modules inside the project can pull in the virtual module as if it were just another library module in the same project:

```
dependencies {
    compile project(':webserver')
}
```

This has the advantage of allowing you to make changes to the library module and have it automatically be pulled in by multiple application projects, without having to fuss with version numbers, artifact repositories, and the like. However, in practice, most production-grade apps would benefit from having the library itself be versioned, as that helps decouple the release schedule of the library from the release schedules of the apps using that library. Use this technique only for lightweight experimental projects... like book sample apps.

Seeing the Reuse In Action

This reusable Web server module will be reused in a few sample projects, including:

- Having a Web app in debug builds to give you information about your running app without disturbing the app itself
- Using a Web app for viewing the output of screenshots taken by an app

Miscellaneous Network Topics

This chapter is a catch-all for various Android capabilities related to network I/O and the Internet, beyond what is covered elsewhere in the book.

(yes, this chapter could have a more exciting rationale for existing, but the author is subject to "Truth in Advertising" laws...)

Prerequisites

Readers of this chapter should have read the core chapters of the book.

Downloading Files

Android 2.3 introduced a `DownloadManager`, designed to handle a lot of the complexities of downloading larger files, such as:

1. Determining whether the user is on WiFi or mobile data, and if so, whether the download should occur
2. Handling when the user, previously on WiFi, moves out of range of the access point and "fails over" to mobile data
3. Ensuring the device stays awake while the download proceeds

`DownloadManager` itself is less complicated than the alternative of writing all of that stuff yourself. However, it does present a few challenges. In this section, we will examine the `Internet/Download` sample project, one that uses `DownloadManager`.

The Permissions

To use `DownloadManager`, you will need to hold the `INTERNET` permission. You will also need the `WRITE_EXTERNAL_STORAGE` permission, as `DownloadManager` can only download to external storage. Note that you need to hold `WRITE_EXTERNAL_STORAGE` even if you are trying to have `DownloadManager` write to some location where that permission might not be needed (e.g., `getExternalFilesDir()` on an Android 4.4+ device). `DownloadManager` is requiring you to hold that permission, more so than the Android framework, and `DownloadManager` requires that permission for all API levels at the present time.

For example, here is the manifest for the `Internet/Download` application, where we request these two permissions:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.downmgr"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true" />

  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".DownloadDemo"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Internet/Download/app/src/main/AndroidManifest.xml)

`WRITE_EXTERNAL_STORAGE` is a `dangerous` permission. With a `targetSdkVersion` of 23 or higher, we need to handle that in our app. This app uses the same `AbstractPermissionActivity` seen in the chapter on permissions, so we can request `WRITE_EXTERNAL_STORAGE` from the user on the first run of our app from the `DownloadDemo` activity:

```
package com.commonsware.android.downmgr;

import android.Manifest;
import android.app.Activity;
import android.app.DownloadManager;
import android.content.Intent;
import android.os.Bundle;
import android.os.StrictMode;
import android.widget.Toast;

public class DownloadDemo extends AbstractPermissionActivity {

  @Override
  protected String[] getDesiredPermissions() {
    return(new String[] {Manifest.permission.WRITE_EXTERNAL_STORAGE});
  }

  @Override
  protected void onPermissionDenied() {
    Toast
      .makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG)
      .show();
    finish();
  }

  @Override
  public void onReady(Bundle savedInstanceState) {
    StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                               .detectNetwork()
                               .penaltyDeath()
                               .build());

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new DownloadFragment()).commit();
    }
  }
```

```
  public void viewLog() {
    startActivity(new Intent(DownloadManager.ACTION_VIEW_DOWNLOADS));
  }
}
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadDemo.java)

That activity then goes on to display a `DownloadFragment`, where most of our code resides.

The Layout

Our sample application has a simple layout, consisting of three buttons:

1. One to kick off a download
2. One to query the status of a download
3. One to display a system-supplied activity containing the roster of downloaded files

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button
    android:id="@+id/start"
    android:text="@string/start_download"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
  />
  <Button
    android:id="@+id/query"
    android:text="@string/query_status"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:enabled="false"
  />
  <Button android:id="@+id/view"
    android:text="@string/view_log"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
```

```
  />
</LinearLayout>
```

(from Internet/Download/app/src/main/res/layout/main.xml)

Requesting the Download

To kick off a download, we first need to get access to the `DownloadManager`. This is a so-called "system service". You can call `getSystemService()` on any activity (or other `Context`), provide it the identifier of the system service you want, and receive the system service object back. However, since `getSystemService()` supports a wide range of these objects, you need to cast it to the proper type for the service you requested.

So, for example, here is the `onCreateView()` method of the `DownloadFragment`, in which we get the `DownloadManager`:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                         Bundle savedInstanceState) {
  mgr=
      (DownloadManager)getActivity().getSystemService(Context.DOWNLOAD_SERVICE);

  View result=inflater.inflate(R.layout.main, parent, false);

  query=result.findViewById(R.id.query);
  query.setOnClickListener(this);
  start=result.findViewById(R.id.start);
  start.setOnClickListener(this);

  result.findViewById(R.id.view).setOnClickListener(this);

  return(result);
}
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadFragment.java)

Most of these managers have no `close()` or `release()` or `goAwayPlease()` sort of methods — you can just use them and let garbage collection take care of cleaning them up.

Given the manager, we can now call an `enqueue()` method to request a download. The name is relevant — do not assume that your download will begin immediately, though often times it will. The `enqueue()` method takes a `DownloadManager.Request` object as a parameter. The `Request` object uses the builder pattern, in that most methods return the `Request` itself, so you can chain a series of calls together with less typing.

For example, the top-most button in our layout is tied to a `startDownload()` method in `DownloadFragment`, shown below:

```
private void startDownload(View v) {
  Uri uri=Uri.parse("https://commonsware.com/misc/test.mp4");

  Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS)
             .mkdirs();

  DownloadManager.Request req=new DownloadManager.Request(uri);

  req.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_WIFI
                             | DownloadManager.Request.NETWORK_MOBILE)
     .setAllowedOverRoaming(false)
     .setTitle("Demo")
     .setDescription("Something useful. No, really.")
     .setDestinationInExternalPublicDir(Environment.DIRECTORY_DOWNLOADS,
                                        "test.mp4");

  lastDownload=mgr.enqueue(req);

  v.setEnabled(false);
  query.setEnabled(true);
}
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadFragment.java)

We are downloading a sample MP4 file, and we want to download it to the external storage area. To do the latter, we are using `getExternalStoragePublicDirectory()` on `Environment`, which gives us a directory suitable for storing a certain class of content. In this case, we are going to store the download in the `Environment.DIRECTORY_DOWNLOADS`, though we could just as easily have chosen `Environment.DIRECTORY_MOVIES`, since we are downloading a video clip. Note that the `File` object returned by `getExternalStoragePublicDirectory()` may point to a not-yet-created directory, which is why we call `mkdirs()` on it, to ensure the directory exists.

We then create the `DownloadManager.Request` object, with the following attributes:

1. We are downloading the specific URL we want, courtesy of the `Uri` supplied to the `Request` constructor
2. We are willing to use either mobile data or WiFi for the download (`setAllowedNetworkTypes()`), but we do not want the download to incur roaming charges (`setAllowedOverRoaming()`)
3. We want the file downloaded as `test.mp4` in the downloads area on the external storage (`setDestinationInExternalPublicDir()`)

We also provide a name (`setTitle()`) and description (`setDescription()`), which are used as part of the notification drawer entry for this download. The user will see these when they slide down the drawer while the download is progressing.

The `enqueue()` method returns an ID of this download, which we hold onto for use in querying the download status.

Keeping Track of Download Status

If the user presses the Query Status button, we want to find out the details of how the download is progressing. To do that, we can call `query()` on the `DownloadManager`. The `query()` method takes a `DownloadManager.Query` object, describing what download(s) you are interested in. In our case, we use the value we got from the `enqueue()` method when the user requested the download:

```
private void queryStatus(View v) {
  Cursor c=
      mgr.query(new DownloadManager.Query().setFilterById(lastDownload));

  if (c == null) {
    Toast.makeText(getActivity(), R.string.download_not_found,
                   Toast.LENGTH_LONG).show();
  }
  else {
    c.moveToFirst();

    Log.d(getClass().getName(),
          "COLUMN_ID: "
              + c.getLong(c.getColumnIndex(DownloadManager.COLUMN_ID)));
    Log.d(getClass().getName(),
          "COLUMN_BYTES_DOWNLOADED_SO_FAR: "
              + c.getLong(c.getColumnIndex(DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR)));
    Log.d(getClass().getName(),
          "COLUMN_LAST_MODIFIED_TIMESTAMP: "
              + c.getLong(c.getColumnIndex(DownloadManager.COLUMN_LAST_MODIFIED_TIMESTAMP)));
    Log.d(getClass().getName(),
          "COLUMN_LOCAL_URI: "
              + c.getString(c.getColumnIndex(DownloadManager.COLUMN_LOCAL_URI)));
    Log.d(getClass().getName(),
          "COLUMN_STATUS: "
              + c.getInt(c.getColumnIndex(DownloadManager.COLUMN_STATUS)));
    Log.d(getClass().getName(),
          "COLUMN_REASON: "
              + c.getInt(c.getColumnIndex(DownloadManager.COLUMN_REASON)));

    Toast.makeText(getActivity(), statusMessage(c), Toast.LENGTH_LONG)
         .show();

    c.close();
  }
}
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadFragment.java)

The `query()` method returns a `Cursor`, containing a series of columns representing the details about our download. There is a series of constants on the `DownloadManager` class outlining what is possible. In our case, we retrieve (and dump to Logcat):

1. The ID of the download (`COLUMN_ID`)
2. The amount of data that has been downloaded to date (`COLUMN_BYTES_DOWNLOADED_SO_FAR`)
3. What the last-modified timestamp is on the download (`COLUMN_LAST_MODIFIED_TIMESTAMP`)
4. Where the file is being saved to locally (`COLUMN_LOCAL_URI`)
5. What the actual status is (`COLUMN_STATUS`)
6. What the reason is for that status (`COLUMN_REASON`)

Note that `COLUMN_LOCAL_URI` may be unavailable, if the user has deleted the downloaded file between when the download completed and the time you try to access the column.

There are a number of possible status codes (e.g., `STATUS_FAILED`, `STATUS_SUCCESSFUL`, `STATUS_RUNNING`). Some, like `STATUS_FAILED`, may have an accompanying reason to provide more details.

Note that you really should close this `Cursor` when you are done with it. `StrictMode`, for example, will complain if you do not.

Download Broadcasts

To find out about the results of the download, we need to register a `BroadcastReceiver`, to watch for two actions used by `DownloadManager`:

1. `ACTION_DOWNLOAD_COMPLETE`, to let us know when the download is done
2. `ACTION_NOTIFICATION_CLICKED`, to let us know if the user taps on the `Notification` displayed on the user's device related to our download

So, in `onResume()` of our fragment, we register a single `BroadcastReceiver` for both of those events:

```
@Override
public void onResume() {
  super.onResume();

  IntentFilter f=
```

```
        new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);

    f.addAction(DownloadManager.ACTION_NOTIFICATION_CLICKED);

    getActivity().registerReceiver(onEvent, f);
  }
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadFragment.java)

That `BroadcastReceiver` is unregistered in `onPause()`:

```
  @Override
  public void onPause() {
    getActivity().unregisterReceiver(onEvent);

    super.onPause();
  }
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadFragment.java)

The `BroadcastReceiver` implementation examines the action string of the incoming `Intent` (via a call to `getAction()` and either displays a `Toast` (for `ACTION_NOTIFICATION_CLICKED`) or enables the start-download `Button`:

```
    public void onReceive(Context ctxt, Intent i) {
      if (DownloadManager.ACTION_NOTIFICATION_CLICKED.equals(i.getAction())) {
        Toast.makeText(ctxt, R.string.hi, Toast.LENGTH_LONG).show();
      }
      else {
        start.setEnabled(true);
      }
    }
  };
}
```

(from Internet/Download/app/src/main/java/com/commonsware/android/downmgr/DownloadFragment.java)

What the User Sees

The user, upon launching the application, sees our three pretty buttons:

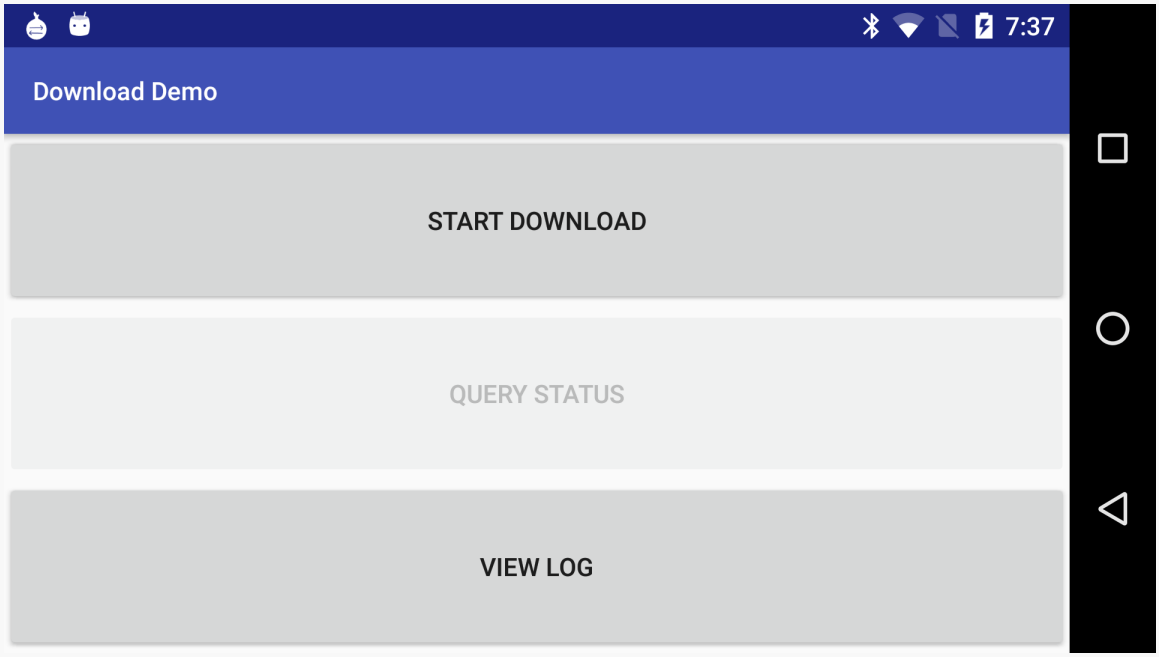


Figure 811: The Download Demo Sample, As Initially Launched

Clicking the first disables the button while the download is going on, and a download icon appears in the status bar (though it is a bit difficult to see, given the poor contrast between Android's icon and Android's status bar):

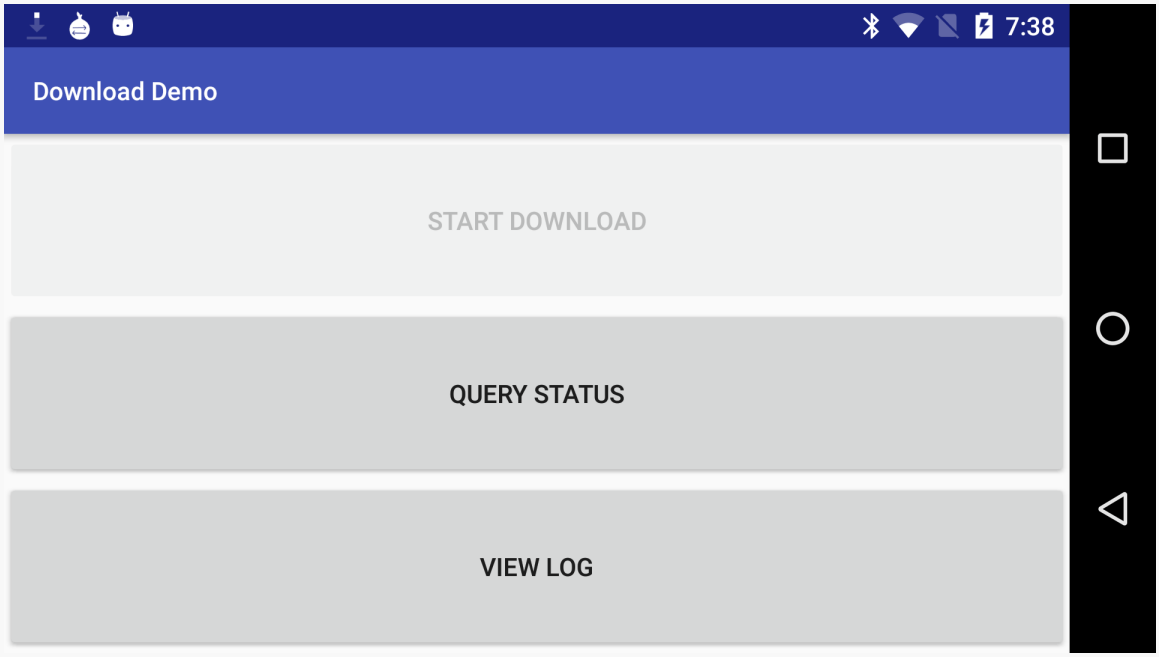


Figure 812: The Download Demo Sample, Downloading

Sliding down the notification drawer shows the user the progress in the form of a `ProgressBar` widget:

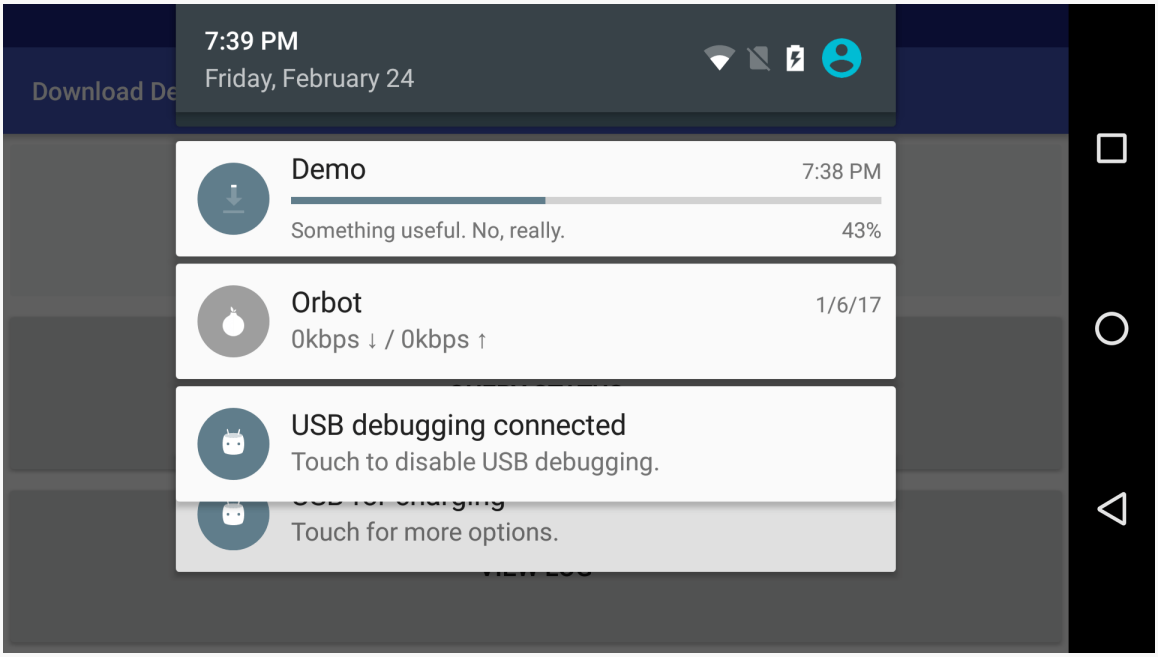


Figure 813: The DownloadManager Notification

Tapping on the entry in the notification drawer returns control to our original activity, where they see a `Toast`, raised by our `BroadcastReceiver`.

If they tap the middle button during the download, a different `Toast` will appear indicating that the download is in progress:

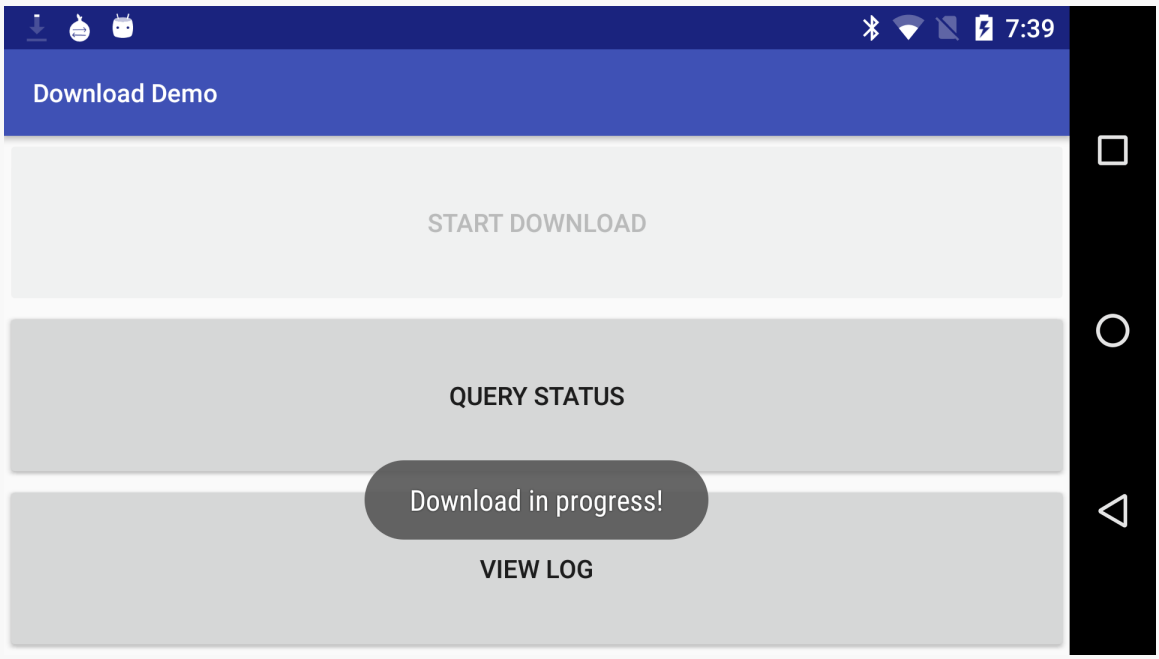


Figure 814: The Download Demo, Showing Download Status

Additional details are also dumped to Logcat:

```
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_ID: 12
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_BYTES_DOWNLOADED_SO_FAR: 615400
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LAST_MODIFIED_TIMESTAMP: 1291988696232
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
12-10 08:45:01.299: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_STATUS: 2
12-10 08:45:01.299: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_REASON: 0
```

Once the download is complete, tapping the middle button will indicate that the download is, indeed, complete, and final information about the download is emitted to Logcat:

```
12-10 08:49:27.360: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_ID: 12
12-10 08:49:27.360: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_BYTES_DOWNLOADED_SO_FAR: 6219229
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
```

```
COLUMN_LAST_MODIFIED_TIMESTAMP: 1291988713409
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_STATUS: 8
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_REASON: 0
```

Tapping the bottom button brings up the activity displaying all downloads, including both successes and failures:

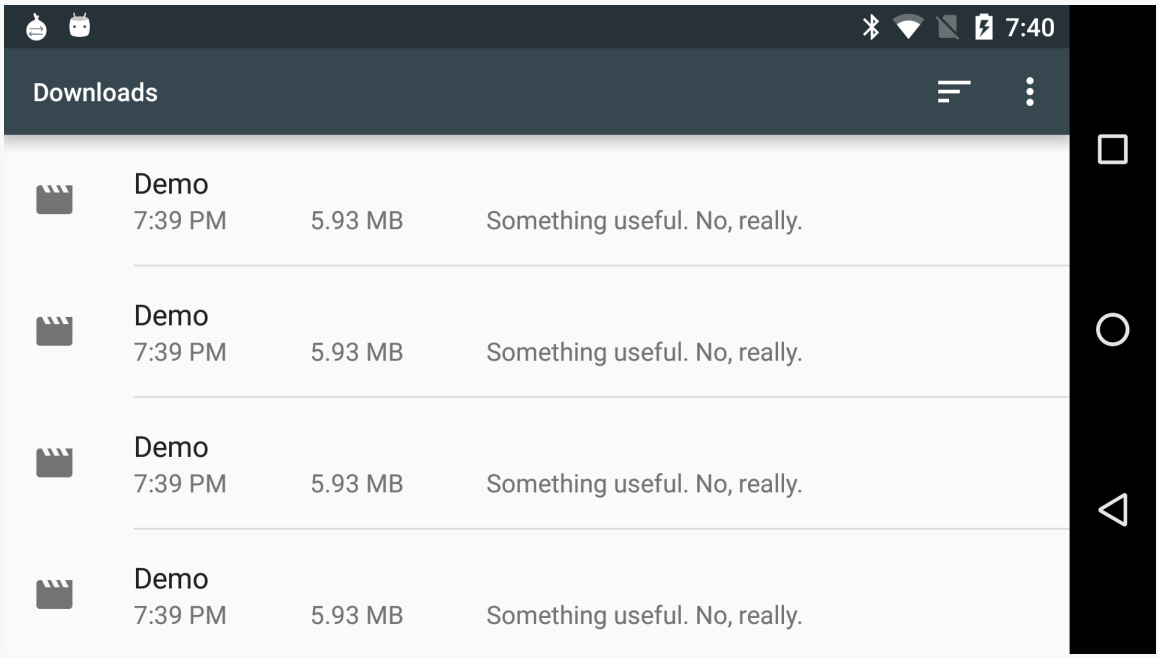


Figure 815: The DownloadManager Results

And, of course, the file is downloaded.

Limitations

While `DownloadManager` nowadays supports HTTPS (SSL) URLs, that was not the case when it was introduced back in Android 2.3. You will want to test any HTTPS URLs you intend to use with `DownloadManager` if you are supporting older versions of Android.

If you display the list of all downloads, and your download is among them, it is a really good idea to make sure that some activity (perhaps one of yours) is able to

respond to an `ACTION_VIEW` `Intent` on that download's MIME type. Otherwise, when the user taps on the entry in the list, they will get a `Toast` indicating that there is nothing available to view the download. This may confuse users. Alternatively, use `setVisibleInDownloadsUi()` on your request, passing in `false`, to suppress it from this list.

Also, starting with Android 5.0, the Downloads app that provides the core implementation of `DownloadManager` keeps track of when other apps get uninstalled. At that point, the Downloads app deletes the files downloaded by `DownloadManager` on behalf of that app. This includes files stored in common locations (e.g., `DIRECTORY_DOWNLOADS`) that would ordinarily survive an uninstall. For example, if you run the `Internet/Download` sample app on an Android 5.0+ device, then uninstall the app, the downloaded file vanishes from the Downloads app. If you elect to use `DownloadManager`, you should either:

- Download the file to a temporary spot, then move it to a long-term location yourself, or
- Advise the user that the file will be deleted if the user uninstalls your app, suggesting that the user might want to make a safe copy of the file

Data Saver

Android has had a per-app "data saver" mode for some time, with an eye towards reducing bandwidth consumption when the device is using a known metered data plan. Android 7.0+ extended this to a device-wide setting,

Apps can be in one of three states as a result:

- The device is normal
- The device is in data-saver mode
- The device is in data-saver mode, but your app is whitelisted by the user

The idea is that if the device is in normal mode, you can do what you want. If the device is in data-saver mode, you should restrict your bandwidth, even if the user whitelists you. Apps that are not whitelisted have no network access while in the background.

To that end, `ConnectivityManager` has three things for you.

First, `isActiveNetworkMetered()` will return `true` if the device is on a metered data connection, `false` otherwise. This has been around for years (API Level 16+), but has not been all that popular, apparently.

Second, Android 7.0 has a `getRestrictBackgroundStatus()` method on `ConnectivityManager`. This returns an `int` that resolves to one of three values:

- `RESTRICT_BACKGROUND_STATUS_DISABLED`
- `RESTRICT_BACKGROUND_STATUS_ENABLED`
- `RESTRICT_BACKGROUND_STATUS_WHITELISTED`

If `isActiveNetworkMetered()` is `true`, and `getRestrictBackgroundStatus()` returns `RESTRICT_BACKGROUND_STATUS_ENABLED`, any attempts to use the network may fail, and so your app should plan accordingly.

If you want to try to react in real-time to changes in the data-saver configuration, you can register a receiver for `ACTION_RESTRICT_BACKGROUND_CHANGED` (defined on `ConnectivityManager`). This will be broadcast for any change in data-saver settings, which means that *your* app's state may not have changed. You will need to call `getRestrictBackgroundStatus()` to find out your current state. Also note that this broadcast is only sent to receivers registered dynamically, via `registerReceiver()`. You cannot register for this broadcast in the manifest.

To try to get on the whitelist, you might be tempted to try using `ACTION_IGNORE_BACKGROUND_DATA_RESTRICTIONS_SETTINGS` to lead the user to add your app to the Data Saver whitelist, so you have normal background network access. However, bear in mind that Google has a similar feature for the battery saver whitelist... and trying to use *that* action got apps banned from the Play Store. At the moment, there is no similar language around the use of the data saver whitelist... but, then again, they did not tell you they were going to ban you for asking to be on the battery saver whitelist until after Android 6.0 shipped.

Trail: Media

Audio Playback

Whether it comes in the form of simple beeps or in the form of symphonies (or gangster rap or whatever), Android applications often need to play audio. A few things in Android can play audio automatically, such as `a Notification`. However, once you get past those, you are on your own.

Fortunately for you, Android offers support for audio playback, and we will examine some of the options in this chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Get Your Media On

In Android, you have a few different places you can pull media clips from — one of these will hopefully fit your needs:

- You can package audio clips as raw resources (`res/raw/` in your project), so they are bundled with your application. The benefit is that you're guaranteed the clips will be there; the downside is that they cannot be replaced without upgrading the application.
- You can package audio clips as assets (`assets/` in your project) and reference them via `file:///android_asset/` URLs in a `Uri`. The benefit over raw resources is that this location works with APIs that expect `Uri` parameters instead of resource IDs. The downside — assets are only replaceable when

the application is upgraded — remains. On the whole, the audio APIs tend to favor raw resources over assets.
- You can download and store media in internal storage, external storage, or possibly on removable storage.
- You can, in some cases, stream media off the Internet, bypassing any local storage.

MediaPlayer for Audio

If you want to play back music, particularly material in MP3 format, you will want to use the `MediaPlayer` class. With it, you can feed it an audio clip, start/stop/pause playback, and get notified on key events, such as when the clip is ready to be played or is done playing.

You have three ways to set up a `MediaPlayer` and tell it what audio clip to play:

- If the clip is a raw resource, use `MediaPlayer.create()` and provide the resource ID of the clip.
- If you have a `Uri` to the clip, use the `Uri`-flavored version of `MediaPlayer.create()`.
- If you have a string path to the clip, just create a `MediaPlayer` using the default constructor, then call `setDataSource()` with the path to the clip. However, in this case, you also need to call `prepare()` or `prepareAsync()`. Both will set up the clip to be ready to play, such as fetching the first few seconds off the file or stream. The `prepare()` method is synchronous; as soon as it returns, the clip is ready to play. The `prepareAsync()` method is asynchronous.

Once the clip is prepared, `start()` begins playback, `pause()` pauses playback (with `start()` picking up playback where `pause()` paused), and `stop()` ends playback. One caveat: you cannot simply call `start()` again on the `MediaPlayer` once you have called `stop()` — we'll cover a workaround a bit later in this section.

To see this in action, take a look at the `Media/Audio` sample project. It contains a single activity — `MainActivity` – that offers playback of a Creative Commons-licensed audio clip, stored as an Ogg Vorbis file as a raw resource (`R.raw.clip`).

In `onCreate()`, we use the static `create()` factory method on `MediaPlayer` to set up a `MediaPlayer` for our audio clip:

```
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    try {
      mp=MediaPlayer.create(this, R.raw.clip);
      mp.setOnCompletionListener(this);
    }
    catch (Exception e) {
      goBlooey(e);
    }
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

We also register the activity itself as the `OnCompletionListener`, to find out if the audio clip is played through to the end.

Under the covers, `create()` not only creates an instance of `MediaPlayer` and sets up the data source, but it also calls `prepare()`, so the `MediaPlayer` is ready for use once `create()` returns. However, this also means that there might be an exception, such as providing an invalid resource ID (e.g., one pointing to a Photoshop file instead of an audio file). `goBlooey()` simply logs the exception and shows a `Toast` as a crude form of error handling:

```
  private void goBlooey(Exception e) {
    Log.e(getClass().getSimpleName(), getString(R.string.msg_error),
      e);
    Toast
      .makeText(this, R.string.msg_error_toast, Toast.LENGTH_LONG)
      .show();
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

Our UI is purely a set of action bar items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/play"
    android:icon="@drawable/ic_play_arrow"
    android:showAsAction="ifRoom"
    android:title="@string/menu_play" />
  <item
    android:id="@+id/pause"
```

```
    android:icon="@drawable/ic_pause"
    android:showAsAction="ifRoom"
    android:title="@string/menu_pause"
    android:visible="false" />
  <item
    android:id="@+id/stop"
    android:icon="@drawable/ic_stop"
    android:showAsAction="ifRoom"
    android:title="@string/menu_stop"
    android:visible="false" />
</menu>
```

(from Media/Audio/app/src/main/res/menu/actions.xml)

Note that only the `play` action bar item is visible at the outset. We will toggle the visibility of the action bar items based on the status of the playback of the clip.

We populate the action bar in `onCreateOptionsMenu()`, also retrieving the three `MenuItem` objects for our three action bar items:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);
    play=menu.findItem(R.id.play);
    pause=menu.findItem(R.id.pause);
    stop=menu.findItem(R.id.stop);

    return(super.onCreateOptionsMenu(menu));
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

`onOptionsItemSelected()` merely delegates the three action items to three similarly-named methods: `play()`, `pause()`, and `stop()`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.play:
        play();
        return (true);

      case R.id.pause:
        pause();
        return (true);

      case R.id.stop:
```

```
      stop();
      return (true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

The `play()` method calls `start()` to cause the `MediaPlayer` to begin playing back the audio clip. It also toggles the visibility of the action bar items, so the `pause` and `stop` ones are now visible:

```
  private void play() {
    mp.start();

    play.setVisible(false);
    pause.setVisible(true);
    stop.setVisible(true);
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

`play()` is asynchronous, as the audio clip plays back on another system-supplied thread. We are not tying up the main application thread by playing back this clip.

`pause()` is similar: it calls `pause()` to cause the `MediaPlayer` to pause playback of the audio clip. It also toggles the visibility of the action bar items, so the `play` and `stop` ones are now visible:

```
  private void pause() {
    mp.pause();

    play.setVisible(true);
    pause.setVisible(false);
    stop.setVisible(true);
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

Where things get a bit complicated is in the `stop()` method. There, we not only want to stop playback, but also set up the `MediaPlayer` to be able to play back from the beginning of the clip:

```
  private void stop() {
    mp.stop();
```

```
    pause.setVisible(false);
    stop.setVisible(false);

    findViewById(android.R.id.content).postDelayed(new Runnable() {
      @Override
      public void run() {
        try {
          mp.prepare();
          mp.seekTo(0);
          play.setVisible(true);
        }
        catch (Exception e) {
          goBlooey(e);
        }
      }
    }, 100);
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

Stopping the playback is merely a matter of calling `stop()` on the `MediaPlayer`. After that, we hide all the action bar items.

To reset the `MediaPlayer` to play the clip again, you need to re-prepare the player, via `prepare()` or `prepareAsync()`. You also need to call `seekTo(0)`. `seekTo()` positions playback at a certain number of milliseconds from the beginning of the audio, so `seekTo(0)` repositions the player back to the beginning.

However, if we try doing this work right away, we get odd results, owing to the asynchronous nature of the media playback. We need to let the `stop()` complete its work before we `prepare()` and `seekTo()`. Unfortunately, there is no listener interface for the stop-completed event. So, we fake it, delaying the "rewind" of the clip by 100 milliseconds.

Also, once the `MediaPlayer` is ready again, we enable the `play` action bar item, allowing playback to commence from start, if the user wants.

Finally, in `onDestroy()`, we `stop()` the `MediaPlayer` if it returns `true` from `isPlaying()`, so the playback does not continue after the activity is destroyed:

```
  @Override
  public void onDestroy() {
    super.onDestroy();

    if (mp.isPlaying()) {
```

```
      mp.stop();
    }
  }
```

(from Media/Audio/app/src/main/java/com/commonsware/android/audio/MainActivity.java)

Other Ways to Make Noise

While `MediaPlayer` is the primary audio playback option, particularly for content along the lines of MP3 files, there are other alternatives if you are looking to build other sorts of applications, notably games and custom forms of streaming audio.

SoundPool

The `SoundPool` class's claim to fame is the ability to overlay multiple sounds, and do so in a prioritized fashion, so your application can just ask for sounds to be played and `SoundPool` deals with each sound starting, stopping, and blending while playing.

This may make more sense with an example.

Suppose you are creating a first-person shooter. Such a game may have several sounds going on at any one time:

1. The sound of the wind whistling amongst the trees on the battlefield
2. The sound of the surf crashing against the beach in the landing zone
3. The sound of booted feet crunching on the sand
4. The sound of the character's own panting as the character runs on the beach
5. The sound of orders being barked by a sergeant positioned behind the character
6. The sound of machine gun fire aimed at the character and the character's squad mates
7. The sound of explosions from the gun batteries of the battleship providing suppression fire

And so on.

In principle, `SoundPool` can blend all of those together into a single audio stream for output. Your game might set up the wind and surf as constant background sounds, toggle the feet and panting on and off based on the character's movement, randomly add the barked orders, and tie the gunfire based on actual game play.

In reality, your average smartphone will lack the CPU power to handle all of that audio without harming the frame rate of the game. So, to keep the frame rate up, you tell `SoundPool` to play at most two streams at once. This means that when nothing else is happening in the game, you will hear the wind and surf, but during the actual battle, those sounds get dropped out — the user might never even miss them — so the game speed remains good.

AudioTrack

The lowest-level Java API for playing back audio is `AudioTrack`. It has two main roles:

1. Its primary role is to support streaming audio, where the streams come in some format other than what `MediaPlayer` handles. While `MediaPlayer` can handle RTSP, for example, it does not offer SIP. If you want to create a SIP client (perhaps for a VOIP or Web conferencing application), you will need to convert the incoming data stream to PCM format, then hand the stream off to an `AudioTrack` instance for playback.
2. It can also be used for "static" (versus streamed) bits of sound that you have pre-decoded to PCM format and want to play back with as little latency as possible. For example, you might use this for a game for in-game sounds (beeps, bullets, or "boing"s). By pre-decoding the data to PCM and caching that result, then using `AudioTrack` for playback, you will use the least amount of overhead, minimizing CPU impact on game play and on battery life.

ToneGenerator

If you want your phone to sound like... well... a phone, you can use `ToneGenerator` to have it play back dual-tone multi-frequency (DTMF) tones. In other words, you can simulate the sounds played by a regular "touch-tone" phone in response to button presses. This is used by the Android dialer, for example, to play back the tones when users dial the phone using the on-screen keypad, as an audio reinforcement.

Note that these will play through the phone's earpiece, speaker, or attached headset. They do not play through the outbound call stream. In principle, you might be able to get `ToneGenerator` to play tones through the speaker loud enough to be picked up by the microphone, but this probably is not a recommended practice.

You can create a `ToneGenerator` through its constructor. This takes two parameters:

- the audio stream associated with this audio

- the volume level to apply to the audio

```
final private ToneGenerator beeper=
  new ToneGenerator(AudioManager.STREAM_NOTIFICATION, 100);
```

The stream indication helps with muting; if the user has muted this particular stream, then `ToneGenerator` will wind up not generating a tone.

Then, when you need a beep, you can call `startTone()`, with the identifier of one of the many DTMF tones listed on the `ToneGenerator` class:

```
beeper.startTone(ToneGenerator.TONE_PROP_NACK);
```

Many of the tones have a fixed duration. In that case, `startTone()` will play that tone to completion, or until you call `stopTone()` to interrupt it.

Some tones will repeat indefinitely. There is a second `startTone()` variant that takes a duration in milliseconds that you can use to automatically stop the tone after a particular period of time. Or, you can use `stopTone()` to stop the one.

Audio Recording

Most Android devices have microphones. On such devices, it might be nice to get audio input from those microphones, whether to record locally, process locally (e.g., speech recognition), or to stream out over the Internet (e.g., voice over IP).

Not surprisingly, Android has some capabilities in this area. Also, not surprisingly, there are multiple APIs, with varying mixes of power and complexity, to allow you to capture microphone input. In this chapter, we will examine `MediaRecorder` for recording audio files and `AudioRecord` for raw microphone input.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having read the chapter on audio playback is probably also a good idea. And, for the section on playing back local streams, you will want to have read up on content providers, particularly the chapter on provider patterns.

Recording by Intent

Just as the easiest way to take a picture with the camera is to use the device's built-in camera app, the easiest way to record some audio is to use a built-in activity for it. And, as with using the built-in camera app, the built-in audio recording activity has some significant limitations.

Requesting the built-in audio recording activity is a matter of calling `startActivityForResult()` for a `MediaStore.Audio.Media.RECORD_SOUND_ACTION` action. You can see this in the `Media/SoundRecordIntent` sample project, specifically the `MainActivity`:

```
package com.commonsware.android.soundrecord;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.MediaStore;
import android.widget.Toast;

public class MainActivity extends Activity {
  private static final int REQUEST_ID=1337;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Intent i=new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);

    startActivityForResult(i, REQUEST_ID);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == REQUEST_ID && resultCode == RESULT_OK) {
      Toast.makeText(this, "Recording finished!", Toast.LENGTH_LONG)
           .show();
    }

    finish();
  }
}
```

(from Media/SoundRecordIntent/app/src/main/java/com/commonsware/android/soundrecord/MainActivity.java)

As with a few other sample apps in this book, the `Media/SoundRecordIntent` uses a `Theme.Translucent.NoTitleBar` activity, avoiding its own UI. Instead, in `onCreate()`, we immediately call `startActivityForResult()` for `MediaStore.Audio.Media.RECORD_SOUND_ACTION`. That will bring up a recording activity:

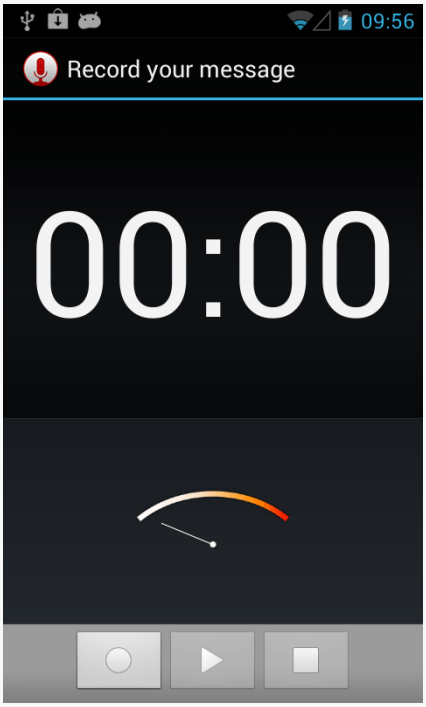


Figure 816: Built-In Sound Recording Activity

If the user records some audio via the "record" `ImageButton` (one with the circle icon) and the "stop" `ImageButton` (one with the square icon), you will get control back in `onActivityResult()`, where you are passed an `Intent` whose `Uri` (via `getData()`) will point to this audio recording in the `MediaStore`.

However:

- You have no control over where the file is stored or what it is named. It appears that, by default, these files are dumped unceremoniously in the root of external storage.
- You have no control over anything about the way the audio is recorded, such as codecs or bitrates. For example, it appears that, by default, the files are recorded in AMR format.
- `ACTION_VIEW` may not be able to play back this audio (leastways, it failed to in testing on a few devices). Whether that is due to codecs, the way the data is put in `MediaStore`, or the limits of the default audio player on Android, is unclear.

Hence, in many cases, while this works, it may not work well enough — or controlled enough — to meet your needs. In that case, you will want to handle the recording yourself, as will be described in the next couple of sections.

Recording to Files

If your objective is to record a voice note, a presentation, or something along those lines, then `MediaRecorder` is probably the class that you want. It will let you specify what sort of media you wish to record, in what format, and to what location. It then handles the actual act of recording.

To illustrate this, let us review the `Media/AudioRecording` sample project.

Our activity's layout consists of a single `ToggleButton` widget named `record`:

```
<ToggleButton xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/record"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:textAppearance="?android:attr/textAppearanceLarge"/>
```

(from Media/AudioRecording/app/src/main/res/layout/activity_main.xml)

Our project is set up to record the output to the `Environment.DIRECTORY_DOWNLOADS` location on external storage. And, if we have a `targetSdkVersion` of 23 or higher (which we do), we need runtime permissions. We also need runtime permissions for `RECORD_AUDIO`, since, well, we are recording audio. So our manifest requests both of those permissions:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.audiorecord"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-permission android:name="android.permission.RECORD_AUDIO"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

  <uses-feature
```

```
    android:name="android.hardware.microphone"
    android:required="true"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/title_activity_main">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Media/AudioRecording/app/src/main/AndroidManifest.xml)

And, following in the pattern demonstrated in the chapter on permissions, we use `AbstractPermissionsActivity` to handle all the details of obtaining our runtime. Our launcher activity — `MainActivity` – inherits from `AbstractPermissionsActivity`, which requests our permissions when the app starts up. `MainActivity` simply overrides the necessary methods. Two of these, `getDesiredPermissions()` and `onPermissionDenied()`, are specifically for the permission logic:

```
  @Override
  protected String[] getDesiredPermissions() {
    return(new String[]{RECORD_AUDIO, WRITE_EXTERNAL_STORAGE});
  }

  @Override
  protected void onPermissionDenied() {
    Toast
      .makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG)
      .show();
    finish();
  }
```

(from Media/AudioRecording/app/src/main/java/com/commonsware/android/audiorecord/MainActivity.java)

`onReady()` serves as our `onCreate()` replacement, and it will be invoked when we have our runtime permissions. There, we load the layout and set the activity itself up as the `OnCheckedChangedListener`, to find out when the user toggles the button:

```
@Override
public void onReady(Bundle savedInstanceState) {
  setContentView(R.layout.activity_main);

  ((ToggleButton)findViewById(R.id.record)).setOnCheckedChangeListener(this);
}
```

(from Media/AudioRecording/app/src/main/java/com/commonsware/android/audiorecord/MainActivity.java)

Also, in `onStart()`, we initialize a `MediaRecorder`, setting the activity up as being the one to handle info and error events about the recording. Similarly, we `release()` the `MediaRecorder` in `onStop()`, to reduce our overhead when we are not in the foreground:

```
@Override
public void onStart() {
  super.onStart();

  recorder=new MediaRecorder();
  recorder.setOnErrorListener(this);
  recorder.setOnInfoListener(this);
}

@Override
public void onStop() {
  recorder.release();
  recorder=null;

  super.onStop();
}
```

(from Media/AudioRecording/app/src/main/java/com/commonsware/android/audiorecord/MainActivity.java)

Most of the work occurs in `onCheckedChanged()`, where we get control when the user toggles the button. If we are now checked, we begin recording; if not, we stop the previous recording:

```
@Override
public void onCheckedChanged(CompoundButton buttonView,
                             boolean isChecked) {
  if (isChecked) {
    output=
      new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS),
        BASENAME);
```

```
    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    recorder.setOutputFile(output.getAbsolutePath());

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD_MR1) {
      recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
      recorder.setAudioEncodingBitRate(160 * 1024);
    }
    else {
      recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    }

    recorder.setAudioChannels(2);

    try {
      recorder.prepare();
      recorder.start();
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
            "Exception in preparing recorder", e);
      Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
    }
  }
  else {
    try {
      recorder.stop();

      MediaScannerConnection
        .scanFile(this, new String[] {output.getAbsolutePath()}, null, null);
    }
    catch (Exception e) {
      Log.w(getClass().getSimpleName(),
            "Exception in stopping recorder", e);
      // can fail if start() failed for some reason
    }

    recorder.reset();
  }
}
```

(from Media/AudioRecording/app/src/main/java/com/commonsware/android/audiorecord/MainActivity.java)

To record audio, we:

- Create a `File` object representing where the recording should be stored, in this case using `Environment.getExternalStoragePublicDirectory()` to find a location on external storage
- Tell the `MediaRecorder` that we wish to record from the microphone, through a call to `setAudioSource()`, that we wish to record a 3GP file via a call to `setOutputFormat()`, and that we wish to record the results to our `File` via a call to `setOutputFile()`

- If we are running on Android 2.3.3 or higher, we can also configure our encoder to be AAC via `setAudioEncoder()` and set our requested bitrate to 160Kbps via `setAudioEncodingBitRate()` — otherwise, we use `setAudioEncoder()` to request AMR narrowband
- Indicate how many audio channels we want via `setAudioChannels()`, such as 2 to attempt to record in stereo
- Kick off the actual recording via calls to `prepare()` (to set up the output file) and `record()`

Stopping the recording, when the user toggles off the button, is merely a matter of calling `stop()` on the `MediaRecorder`, then using `MediaScannerConnection` to get the resulting file indexed by the `MediaStore`, so it shows up for desktop users, media apps, etc.

Because we told the `MediaRecorder` that our activity was our `OnErrorListener` and `OnInfoListener`, we have to implement those interfaces on the activity and implement their required methods (`onError()` and `onInfo()`, respectively). In the normal course of events, neither of these should be triggered. If they are, we are passed an `int` value (typically named `what`) that indicates what happened:

```
@Override
public void onInfo(MediaRecorder mr, int what, int extra) {
  String msg=getString(R.string.strange);

  switch (what) {
    case MediaRecorder.MEDIA_RECORDER_INFO_MAX_DURATION_REACHED:
      msg=getString(R.string.max_duration);
      break;

    case MediaRecorder.MEDIA_RECORDER_INFO_MAX_FILESIZE_REACHED:
      msg=getString(R.string.max_size);
      break;
  }

  Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
}

@Override
public void onError(MediaRecorder mr, int what, int extra) {
  Toast.makeText(this, R.string.strange, Toast.LENGTH_LONG).show();
}
```

(from Media/AudioRecording/app/src/main/java/com/commonsware/android/audiorecord/MainActivity.java)

Here, we just raise a `Toast` in either case, with either a generic message or a specific message for the cases where the maximum time duration or the maximum file size for our recording has been reached.

The results are that we get a recording on external storage (typically in a `Downloads` directory) after we toggle the button on, record some audio, then toggle the button off.

`MediaRecorder` is rather fussy about the order of method calls for its configuration. For example, you must call `setAudioEncoder()` *after* the call to `setOutputFormat()`.

Also, the available codecs and file types are rather limited. Notably, Android lacks the ability to record to MP3 format, perhaps due to patent licensing issues.

Recording to Streams

The nice thing about recording to files is that Android handles all of the actual file I/O for us. The downside is that because Android handles all of the actual file I/O for us, it can only write files that are accessible to it and our process, meaning external storage. This may not be suitable in all cases, such as wanting to record to some form of private encrypted storage.

The good news is that Android does support recording to streams, in the form of a pipe created by `ParcelFileDescriptor` and `createPipe()`. This follows the same basic pattern that we saw in the chapter on content provider patterns, where we served a stream via a pipe. However, as you will see, there are some limits on how well we can do this.

To demonstrate and explain, let us examine the `Media/AudioRecordStream` sample project. This is nearly a complete clone of the previous sample, so we will only focus on the changes in this section.

The author would like to thank Lucio Maciel for his assistance in getting this example to work.

Setting Up the Stream

The biggest change, by far, is in our `setOutputFile()` call. Before, we supplied a path to external storage. Now, we supply the write end of a pipe:

```
      recorder.setOutputFile(getStreamFd());
```

(from Media/AudioRecordStream/app/src/main/java/com/commonsware/android/audiorecstream/MainActivity.java)

Our `getStreamFd()` method looks a lot like the `openFile()` method of our pipe-providing provider:

```
private FileDescriptor getStreamFd() {
  ParcelFileDescriptor[] pipe=null;

  try {
    pipe=ParcelFileDescriptor.createPipe();

    new TransferThread(new AutoCloseInputStream(pipe[0]),
                       new FileOutputStream(getOutputFile())).start();
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(), "Exception opening pipe", e);
  }

  return(pipe[1].getFileDescriptor());
}
```

(from Media/AudioRecordStream/app/src/main/java/com/commonsware/android/audiorecstream/MainActivity.java)

We create our pipe with `createPipe()`, spawn a `TransferThread` to copy the recording from an `InputStream` to a `FileOutputStream`, and return the write end of the pipe. However, `setOutputFile()` on `MediaRecorder` takes the actual integer file descriptor, not a `ParcelFileDescriptor`, so we use `getFileDescriptor()` to retrieve the file descriptor and return that.

Our `TransferThread` is similar to the one from the content provider sample, except that we pass over a `FileOutputStream`, so we can not only `flush()` but also `sync()` when we are done writing:

```
static class TransferThread extends Thread {
  InputStream in;
  FileOutputStream out;

  TransferThread(InputStream in, FileOutputStream out) {
    this.in=in;
    this.out=out;
  }

  @Override
  public void run() {
```

```
      byte[] buf=new byte[8192];
      int len;

      try {
        while ((len=in.read(buf)) >= 0) {
          out.write(buf, 0, len);
        }

        in.close();

        out.flush();
        out.getFD().sync();
        out.close();
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(),
              "Exception transferring file", e);
      }
    }
  }
```

(from Media/AudioRecordStream/app/src/main/java/com/commonsware/android/audiorecstream/MainActivity.java)

Changes in Recording Configuration

The biggest limitation of a pipe's stream is that it is *purely* a stream. You cannot rewind re-read earlier bits of data. In other words, the stream is not seekable.

That is a problem with `MediaRecorder` in some configurations. For example, a 3GP file contains a header with information about the overall file, information that `MediaRecorder` does not know until the recording is complete. In the case of a file, `MediaRecorder` can simply rewind and update the header with the final data when everything is done. However, that is not possible with a pipe-based stream.

However, some configurations will work, notably "raw" ones that just have the recorded audio, with no type of header. That is what we use in this sample.

Specifically, we now write to a `.amr` file:

```
  private static final String BASENAME="recording-stream.amr";
```

(from Media/AudioRecordStream/app/src/main/java/com/commonsware/android/audiorecstream/MainActivity.java)

We also set our output format to `RAW_AMR`, and our encoder to `AMR_NB`:

```
      recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
      recorder.setOutputFormat(MediaRecorder.OutputFormat.RAW_AMR);
      recorder.setOutputFile(getStreamFd());
      recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
      recorder.setAudioChannels(2);
```

(from Media/AudioRecordStream/app/src/main/java/com/commonsware/android/audiorecstream/MainActivity.java)

This combination works. Other combinations might also work. But our approach of writing the 3GP file, as in the file-based example, will not work.

Raw Audio Input

Just as `AudioTrack` allows you to play audio supplied as raw 8- or 16-bit PCM input, `AudioRecord` allows you to record audio from the microphone, supplied to you in PCM format. It is then up to you to actually do something with the raw byte PCM data, including converting it to some other format and container as needed.

Note that you need `RECORD_AUDIO` to work with `AudioRecord`, just as you need it to work with `MediaRecorder`.

Requesting the Microphone

As noted in the opening paragraph of this chapter, most Android devices have microphones. The key word there is *most*. Not all Android devices will have microphones, as only some tablets (and fewer Android TV devices) will support microphone input.

As with most of this optional hardware, the solution is to use `<uses-feature>`. In that case, you would request the `android.hardware.microphone` feature, with `android:required="false"` if you felt that you do not absolutely *need* a microphone. In that case, you would use `hasSystemFeature()` on `PackageManager` to determine at runtime if you do indeed have a microphone.

Note that the `RECORD_AUDIO` permission implies that you need a microphone. Hence, even if you skip the `<uses-feature>` element, your app will still only ship to devices that have a microphone. If the microphone is optional, be sure to include `android:required="false"`, so your app will be available to devices that lack a microphone.

Video Playback

Just as Android supports audio playback, it also supports video playback of local and streaming content. Unlike audio playback – which supports a mix of high-level and low-level APIs – video playback offers a purely high-level interface, in the form of the same `MediaPlayer` class you used for audio playback. To keep things a bit simpler, though, Android does offer a `VideoView` widget you can drop in an activity or fragment to play back video.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, along with the chapter on audio playback.

Moving Pictures

Video clips get their own widget, the `VideoView`. Put it in a layout, feed it an MP4 video clip, and you get playback!

For example, take a look at this layout, from the `Media/Video` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <VideoView
     android:id="@+id/video"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
```

```
    />
</LinearLayout>
```

(from Media/Video/app/src/main/res/layout/main.xml)

The layout is simply a full-screen video player. Whether it will use the full screen will be dependent on the video clip, its aspect ratio, and whether you have the device (or emulator) in portrait or landscape mode.

Wiring up the Java is almost as simple:

```
package com.commonsware.android.video;

import java.io.File;
import android.Manifest;
import android.app.Activity;
import android.graphics.PixelFormat;
import android.os.Bundle;
import android.os.Environment;
import android.widget.MediaController;
import android.widget.Toast;
import android.widget.VideoView;

public class VideoDemo extends AbstractPermissionActivity {
  private VideoView video;
  private MediaController ctlr;

  @Override
  protected String[] getDesiredPermissions() {
    return(new String[] {Manifest.permission.WRITE_EXTERNAL_STORAGE});
  }

  @Override
  protected void onPermissionDenied() {
    Toast
      .makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG)
      .show();
    finish();
  }

  @Override
  public void onReady(Bundle state) {
    getWindow().setFormat(PixelFormat.TRANSLUCENT);
    setContentView(R.layout.main);

    File clip=new File(Environment.getExternalStorageDirectory(),
                       "test.mp4");
```

```
    if (clip.exists()) {
      video=(VideoView)findViewById(R.id.video);
      video.setVideoPath(clip.getAbsolutePath());

      ctlr=new MediaController(this);
      ctlr.setMediaPlayer(video);
      video.setMediaController(ctlr);
      video.requestFocus();
      video.start();
    }
  }
}
```

(from Media/Video/app/src/main/java/com/commonsware/android/video/VideoDemo.java)

We use the `AbstractPermissionActivity` profiled earlier in the book, as we need `READ_EXTERNAL_STORAGE` rights to be able to read a `test.mp4` video from the root of external storage.

Beyond that, we:

1. Confirm that our video file exists on external storage
2. Tell the `VideoView` which file to play
3. Create a `MediaController` pop-up panel and cross-connect it to the `VideoView`
4. Give the `VideoView` the focus and start playback

The biggest trick with `VideoView` is getting a video clip onto the device. While `VideoView` does support some streaming video, the requirements on the MP4 file are fairly stringent. If you want to be able to play a wider array of video clips, you need to have them on the device, preferably on an SD card.

The crude `VideoDemo` class assumes there is an MP4 file named `test.mp4` in the root of external storage on your device or emulator. Once there, the Java code shown above will give you a working video player:

Figure 817: VideoDemo, Showing a Creative Commons-Licensed Video

Tapping on the video will pop up the playback controls:

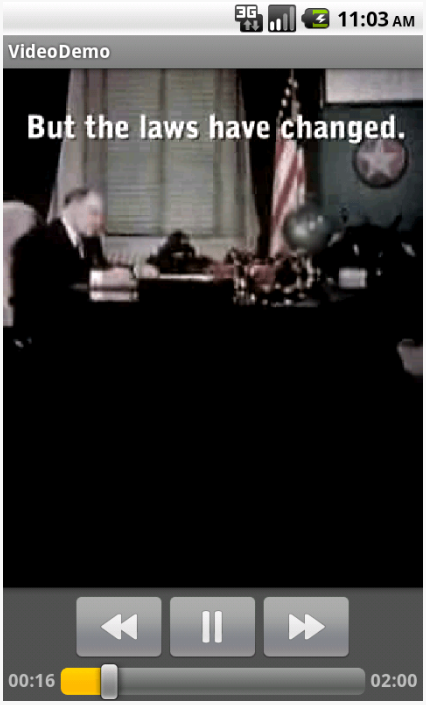
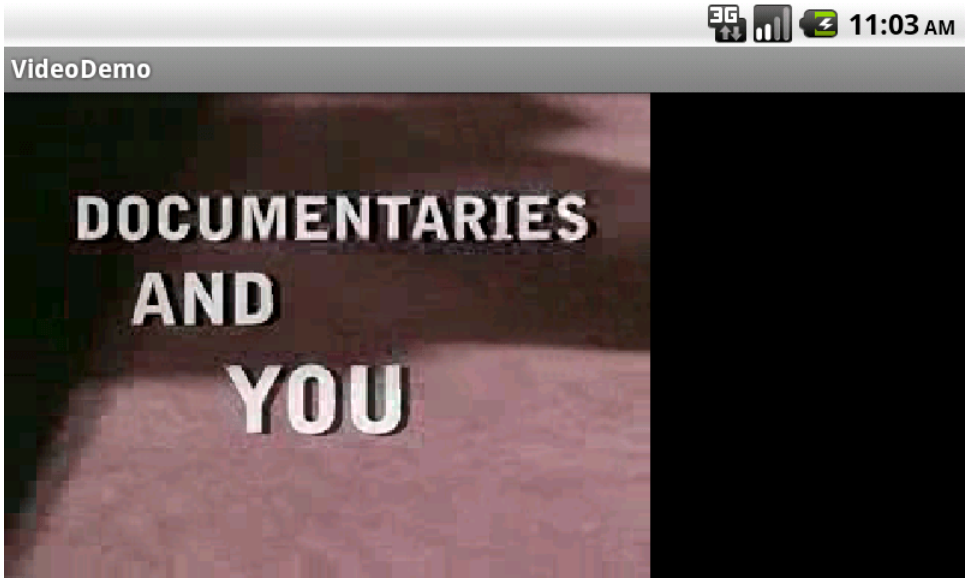


Figure 818: VideoDemo, Showing Media Controls

The video will scale based on space as well:



Using the Camera via 3rd-Party Apps

Most Android devices will have a camera, since they are fairly commonplace on mobile devices these days. You, as an Android developer, can take advantage of the camera, for everything from snapping tourist photos to scanning barcodes. If you wish to let other apps do the "heavy lifting" for you, working with the camera can be fairly straightforward. If you want more control, you can work with the camera directly, though this control comes with greater complexity.

You can also record videos using the camera. Once again, you have the option of either using a third-party activity, or doing it yourself.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the material on implicit `Intents`. You also need to read the chapters on the `ContentProvider` component, particularly the coverage of `FileProvider`.

Being Specific About Features

If your app needs a camera — by any of the means cited in this chapter – you should include a `<uses-feature>` element in the manifest indicating your requirements. However, you need to be fairly specific about your requirements here.

For example, the Nexus 7 (2012) has a camera... but only a front-facing camera. This facilitates apps like video chat. However, the `android.hardware.camera` implies that you need a high-resolution rear-facing camera. Hence, to work with the Nexus 7's camera, you need to:

- Require the `CAMERA` permission (if you are using the `Camera` directly)
- Not require the `android.hardware.camera` feature (`android:required="false"`)
- Optionally require the `android.hardware.camera.front` feature (if your app definitely needs a front-facing camera)

At runtime, you would use `hasSystemFeature()` on `PackageManager`, or interrogate the `Camera` class for available cameras, to determine what you have access to.

Note that if you want to record audio when recording videos, you should also consider the `android.hardware.microphone` feature.

Still Photos: Letting the Camera App Do It

The easiest way to take a picture is to not take the picture yourself, but let somebody else do it. The most common implementation of this approach is to use an `ACTION_IMAGE_CAPTURE` `Intent` to bring up the user's default camera application, and let it take a picture on your behalf.

In theory, this is fairly simple:

- You call `startActivityForResult()` on an `ACTION_IMAGE_CAPTURE` `Intent`
- You either get a thumbnail photo back or — if you provided `EXTRA_OUTPUT` as an extra on the `ACTION_IMAGE_CAPTURE` `Intent` — a full-sized photo should be written to where you designated in `EXTRA_OUTPUT`

In practice, this gets complicated, in part because Android 7.0 is trying to get rid of `file` schemes on `Uri` values. As a result, `EXTRA_OUTPUT` cannot point to a file. Instead, it has to point to a `ContentProvider`, such as `FileProvider`.

To see this in use, take a look at the `Camera/FileProvider` sample project. This app will use system-supplied activities to take a picture, then view the result, without actually implementing any of its own UI.

Setting the Theme

Of course, we still need an activity, so our code can be launched by the user. We just set it up with `Theme.Translucent.NoTitleBar`, so no UI will be created for it:

```
    <activity
      android:name=".MainActivity"
```

```
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
```

(from Camera/FileProvider/app/src/main/AndroidManifest.xml)

Requesting the Feature

Since our app is useless without a camera, we have the `<uses-feature>` element in the manifest stating that we need some sort of camera:

```
  <uses-feature android:name="android.hardware.camera.any" />
```

(from Camera/FileProvider/app/src/main/AndroidManifest.xml)

This will prevent our app from being installed on a device that lacks a camera.

Adding the FileProvider

As noted earlier, Android 7.0 has a ban on `file:Uri` values, if your `targetSdkVersion` is `24` or higher. In particular, you cannot use a `file:Uri` in an `Intent`, whether as the "data" aspect of the `Intent` or as the value of an extra.

The proper way to implement this is to use a `ContentProvider`, such as a `FileProvider`, as is covered in one of the chapters on providers. This is a fair bit more complicated, and not all camera apps will work well with a `content:Uri`, but our options are limited.

Our `res/xml/provider_paths.xml` metadata for the `FileProvider` indicate that we want to serve up the contents of the `photos/` directory inside of `getFilesDir()`, with a `Uri` segment of `/p/` mapping to that location:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
  <files-path
    name="p"
    path="photos" />
</paths>
```

(from Camera/FileProvider/app/src/main/res/xml/provider_paths.xml)

Our manifest now has a `<provider>` element for our `FileProvider` subclass, named `LegacyCompatFileProvider`:

```
    <provider
      android:name="LegacyCompatFileProvider"
      android:authorities="${applicationId}.provider"
      android:exported="false"
      android:grantUriPermissions="true">
      <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths" />
    </provider>
```

(from Camera/FileProvider/app/src/main/AndroidManifest.xml)

That element:

- Provides a pointer to that metadata resource, to configure `FileProvider`
- Uses the `applicationId` of this app as the basis of our `authorities` value, using a manifest placeholder
- Blocks access to third parties (`android:exported="false"`) except where we explicitly grant permission in our Java code (`android:grantUriPermissions="true"`)

`LegacyCompatFileProvider` is the same implementation as from the original discussion of `FileProvider`, using `LegacyCompatCursorWrapper` to help increase the odds that clients of this `ContentProvider` will behave properly:

```
package com.commonsware.android.camcon;

import android.database.Cursor;
import android.net.Uri;
import android.support.v4.content.FileProvider;
import com.commonsware.cwac.provider.LegacyCompatCursorWrapper;

public class LegacyCompatFileProvider extends FileProvider {
  @Override
  public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String
sortOrder) {
    return(new LegacyCompatCursorWrapper(super.query(uri, projection, selection, selectionArgs,
sortOrder)));
  }
}
```

(from Camera/FileProvider/app/src/main/java/com/commonsware/android/camcon/LegacyCompatFileProvider.java)

Now we can start the work of taking pictures.

Taking a Picture

At this point, we can start using the provider, to give us a `Uri` that we can use with `EXTRA_OUTPUT` and an `ACTION_IMAGE_CAPTURE` `Intent`.

Our `onCreate()` method spends a lot of lines to eventually make that `startActivityForResult()` call:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (savedInstanceState==null) {
    output=new File(new File(getFilesDir(), PHOTOS), FILENAME);

    if (output.exists()) {
      output.delete();
    }
    else {
      output.getParentFile().mkdirs();
    }

    Intent i=new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    Uri outputUri=FileProvider.getUriForFile(this, AUTHORITY, output);

    i.putExtra(MediaStore.EXTRA_OUTPUT, outputUri);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
      i.addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
    }
    else if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.JELLY_BEAN) {
      ClipData clip=
        ClipData.newUri(getContentResolver(), "A photo", outputUri);

      i.setClipData(clip);
      i.addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
    }
    else {
      List<ResolveInfo> resInfoList=
        getPackageManager()
          .queryIntentActivities(i, PackageManager.MATCH_DEFAULT_ONLY);

      for (ResolveInfo resolveInfo : resInfoList) {
        String packageName = resolveInfo.activityInfo.packageName;
        grantUriPermission(packageName, outputUri,
          Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
```

```
      }
    }

    try {
      startActivityForResult(i, CONTENT_REQUEST);
    }
    catch (ActivityNotFoundException e) {
      Toast.makeText(this, R.string.msg_no_camera, Toast.LENGTH_LONG).show();
      finish();
    }
  }
  else {
    output=(File)savedInstanceState.getSerializable(EXTRA_FILENAME);
  }
}
```

(from Camera/FileProvider/app/src/main/java/com/commonsware/android/camcon/MainActivity.java)

When we are first run, our `savedInstanceState` `Bundle` will be `null`. If it is not `null`, we know that we are coming back from some prior invocation of this activity, and so we do not need to call `startActivityForResult()` to take a picture.

First, we need a `File` pointing to where we want the photo to be stored. We create a directory inside of `getFilesDir()` (named `photos` via the `PHOTOS` constant), and in there identify a file (named `CameraContentDemo.jpeg` via the `FILENAME` constant).

Then, we create the `ACTION_IMAGE_CAPTURE` `Intent`, use `FileProvider.getUriForFile()` to get a `Uri` pointing to our desired `File`, then put that `Uri` in the `EXTRA_OUTPUT` extra of the `Intent`.

Now, though, we have to grant permissions to be able to write to that `Uri`.

If we are on Android 5.0+, calling `addFlags(FLAG_GRANT_WRITE_URI_PERMISSION)` not only affects the "data" aspect of the `Intent`, but also `EXTRA_OUTPUT`, due to a bit of a hack that Google added to the `Intent` class. So, that scenario is simple. The problem comes in with Android 4.4 and older devices, where `addFlags(FLAG_GRANT_WRITE_URI_PERMISSION)` does not affect `Uri` values passed in extras.

For Android 4.2 through 4.4, we can use a trick: while flags skip over `Intent` extras, flags *do* apply to a `ClipData` that you attach to the `Intent` via `setClipData()`. Even though the camera app will never use this `ClipData`, by wrapping our `Uri` in a `ClipData` and attaching that to the `Intent`, our `addFlags(FLAG_GRANT_WRITE_URI_PERMISSION)` will affect that `Uri`. The fact that the

camera app gets the `Uri` from `EXTRA_OUTPUT`, instead of from the `ClipData`, makes no difference.

For Android 4.1 and older devices, though, there is no means for us to simply indicate on the `Intent` itself that it is fine for the app handling our request to write to our `Uri`. Instead, we:

- Find all activities that support `ACTION_IMAGE_CAPTURE`, using `PackageManager` and `queryIntentActivities()`
- Iterate over all of them and call `grantUriPermission()`, inherited from `Context`, to allow the app to read and write from our `Uri`

This allows our `Intent` to succeed for any camera app... at least those that properly handle `content:` `Uri` values.

Finally, after all of that, we can call `startActivityForResult()`. However, in case the user does not have a camera app, we wrap that call in a `try/catch` block, watching for an `ActivityNotFoundException`.

Saving the State

In order to be able to save the `File` across configuration changes, we stuff it in the saved instance state `Bundle` in `onSaveInstanceState()`:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putSerializable(EXTRA_FILENAME, output);
}
```

(from Camera/FileProvider/app/src/main/java/com/commonsware/android/camcon/MainActivity.java)

This is what allows us to pull that value back out in `onReady()`.

Viewing the Photo

Our `onActivityResult()` method then uses the same `File`, creating an `ACTION_VIEW` `Intent`, pointing at our output `Uri`, granting read permission on that `Uri`, indicating the MIME type is `image/jpeg`, and starting up an activity for that.

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
```

```
                                Intent data) {
  if (requestCode == CONTENT_REQUEST) {
    if (resultCode == RESULT_OK) {
      Intent i=new Intent(Intent.ACTION_VIEW);
      Uri outputUri=FileProvider.getUriForFile(this, AUTHORITY, output);

      i.setDataAndType(outputUri, "image/jpeg");
      i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

      try {
        startActivity(i);
      }
      catch (ActivityNotFoundException e) {
        Toast.makeText(this, R.string.msg_no_viewer, Toast.LENGTH_LONG).show();
      }

      finish();
    }
  }
}
```

(from Camera/FileProvider/app/src/main/java/com/commonsware/android/camcon/MainActivity.java)

We do not have to fuss with the `grantUriPermissions()` loop, as `addFlags()` has always granted permission to the "data" aspect of the `Intent` (our `Uri`).

We wrap the `startActivity()` call in another `try/catch` block, watching for `ActivityNotFoundException`. Not all devices will have an image viewing app that supports a `content` `Uri`. For example, stock Android 5.1 (e.g., on a Nexus 4) will not have such an image viewer.

The Caveats

There are several downsides to this approach.

First, you have no control over the camera app itself. You do not even really know what app it is. You cannot dictate certain features that you would like (e.g., resolution, color effects). You simply blindly ask for a photo and get the result.

Also, since you do not know what the camera app is or behaves like, you cannot document that portion of your application's flow very well. You can say things like "at this point, you can take a picture using your chosen camera app", but that is about as specific as you can get.

As noted above, it is possible that your app's process will be terminated while your app is not in the foreground, because the user is taking a picture using the third-party camera app. Whether or not this happens depends on how much system RAM the camera app uses and what else is all going on with the device. But, it does

happen. Your app should be able to cope with such things, just as we are doing with the saved instance state `Bundle`. However, many developers do not expect their process to be replaced between a call to `startActivityForResult()` and the corresponding `onActivityResult()` callback.

Not every camera app will support a `content` `Uri` for the `EXTRA_OUTPUT` value. In fact, Google's own camera app did not do this until the summer of 2016. With `ACTION_VIEW`, since the `content` `Uri` is in the "data" facet of the `Intent`, the `<intent-filter>` elements in the manifest will ensure that our `Intent` only goes to an activity that advertises support for `content`. However, there is no equivalent of this for `Uri` values in extras. And so we will launch the camera app, which then may crash because it does not like our `Uri`, and our app does not really find out about the problem, other than not getting `RESULT_OK` in `onActivityResult()`.

Finally, some camera apps misbehave, returning odd results, such as a thumbnail-sized image rather than a max-resolution image. There is little you can do about this.

Permissions and Third-Party Camera Apps

The sample app shown above does not request any permissions — there are no `<uses-permission>` elements in the manifest. While there is a `CAMERA` permission to use the camera, we do not need it. The camera app that we are starting needs it.

However, you might need the `CAMERA` permission elsewhere in your app. For example, you might be embedding some third-party scanning library, where it will use the camera directly in your app.

In this case, on Android 6.0+, for apps with a `targetSdkVersion` of 23 or higher, even though `ACTION_IMAGE_CAPTURE` itself does not normally need the `CAMERA` permission, you cannot use `ACTION_IMAGE_CAPTURE` if the user has not granted the `CAMERA` permission at runtime.

In other words:

If you do this...	...can you use `ACTION_IMAGE_CAPTURE`?
If you do not request the `CAMERA` permission	Yes
If you request the `CAMERA` permission, and your `targetSdkVersion` is below 23	Yes

If you do this...	...can you use ACTION_IMAGE_CAPTURE?
If you request the CAMERA permission, and the user grants the runtime permission	Yes
If you request the CAMERA permission, and you have not yet asked the user for the runtime permission	No
If you request the CAMERA permission, and the user **denies** the runtime permission	No

A Matter of Orientation

When you take a picture using an Android device — whether using ACTION_IMAGE_CAPTURE or working with the camera APIs directly, you may find that your picture turns out strange. For example, you might take a picture in portrait mode, then find that some image viewers will show you a portrait picture, while others show you a landscape picture with its contents rotated.

That is due to the way Android camera hardware encodes the JPEG images that it takes. The orientation that you take the picture in may not be the orientation of the result.

EXIF Tags

JPEG images can have EXIF tags. These represent metadata about the image itself. For example, if you hear that an image has been "geotagged", that means that the image has EXIF tags that contain the latitude and longitude of where the picture was taken.

These tags are contained in the JPEG file but are in a separate section from the actual image data itself. Tools can read in the EXIF tags and use them for additional information for the user (e.g., an image viewer with an integrated map to show where the picture was taken).

EXIF Tags and Camera Images

One EXIF tag is the "orientation" tag. In effect, this tag is a message from whatever created the image (e.g., camera hardware) to whatever is showing the camera image, saying "could you please rotate this image for me? #kthxbye".

In other words, the camera hardware is being lazy.

A lot of camera hardware is designed to take landscape images, particularly when using a rear-facing camera, as that is the traditional way that cameras were held by default, going back decades. In an ideal world, if the user took a portrait photo, the camera hardware would take a portrait picture. Or, at least, the camera hardware would take a landscape picture, but then rotate the image to be portrait before delivering the JPEG to whatever app requested the image.

Some camera hardware does just that.

However, other camera hardware leaves the image as a landscape image, regardless of how the device was held when the image was taken. Instead, the camera hardware will set the orientation tag to indicate how image viewers should rotate the image, to reflect what the image really should look like.

EXIF Tags and Android

Of course, this would not be a problem if all image viewers paid attention to the orientation tag. However, many do not, particularly on Android... because `BitmapFactory` ignores all EXIF tags. As a result, you get the unmodified image, instead of one rotated as the camera hardware requested.

And so, if you blindly load the image, it will show up without taking the orientation tag into account.

If *you* want to take the orientation tag into account, you need to find out the value of that tag. `BitmapFactory` will not help you here. However, `ExifInterface` can... though *which* `ExifInterface` you use is important.

`android.media.ExifInterface` that Android used for years has security flaws. Android 7.0+ devices should all ship with a patched version. Some Android 6.0 devices might get a patch. Everything else will go unpatched, and if your app uses `android.media.ExifInterface`, your app may expose the user to security risks.

Fortunately, alternative `ExifInterface` implementations exist, that not only avoid the security flaw, but also support `InputStream` as well as `File`. That is only available on `android.media.ExifInterface` starting with Android 7.0; older versions only supported a `File`, which is awkward in modern Android development.

The simplest solution for most developers would be to use the `exifinterface` artifact from the Android Support Library

```
dependencies {
  implementation 'com.android.support:exifinterface:25.1.0'
}
```

This has the same API as does the Android 7.0+ edition of `ExifInterface`, including `InputStream` support.

However, for whatever reason, the API that Google has elected to expose through both of their supplied `ExifInterface` classes pales in comparison to the EXIF classes that they have elsewhere, such as the AOSP editions of the camera and gallery apps. A version of this code is available as an artifact published by Alessandro Crugnola and is demonstrated in the next section. If your EXIF needs are fairly limited, using the Google-supplied `ExifInterface` classes is simple. But even for something as seemingly simple as rotating an image, you need a more robust EXIF API.

You Spin (Photos) Right Round

The `Camera/EXIFRotater` sample project contains three images in `assets/`, culled from this GitHub repository that supplements this article on the problems with EXIF orientation handling. Specifically, we have images with orientation tag values of 3, 6, and 8, which are the most common ones that you will encounter.

The objective of this app is to show one of those images in its original form and rotated in accordance with the EXIF orientation tag:

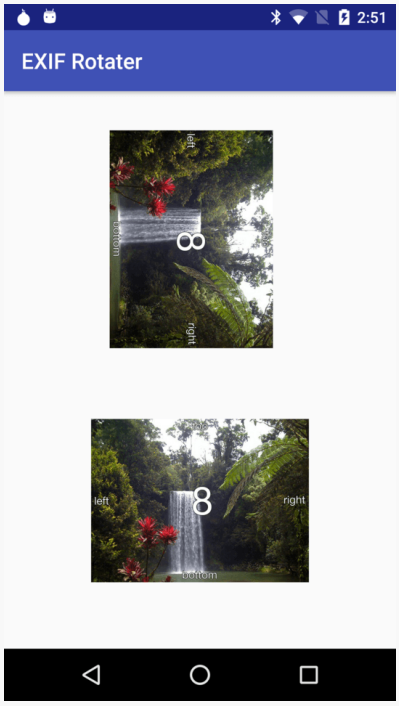


Figure 819: EXIF Rotater Sample App, with Original and Rotated Images

However, there are two product flavors in this project, reflecting two different ways of getting that visual output: rotating the `ImageView` and rotating the image itself. These are controlled via a `ROTATE_BITMAP` value added to `BuildConfig`:

```
    productFlavors {
        image {
            dimension "default"
            buildConfigField "boolean", "ROTATE_BITMAP", "false"
        }
        matrix {
            dimension "default"
            buildConfigField "boolean", "ROTATE_BITMAP", "true"
        }
    }
```

(from Camera/EXIFRotater/app/build.gradle)

The `MainActivity` kicks off an `ImageLoadThread` in `onCreate()`. That thread is responsible for loading (and, if appropriate, rotating) the image. When that is done, the thread will post an `ImageLoadedEvent` to an event bus (using greenrobot's EventBus) to have the UI display the image (and, if needed, rotate a copy of it):

```
private static class ImageLoadThread extends Thread {
  private final Context ctxt;

  ImageLoadThread(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
  }

  @Override
  public void run() {
    AssetManager assets=ctxt.getAssets();

    try {
      InputStream is=assets.open(ASSET_NAME);
      ExifInterface exif=new ExifInterface();

      exif.readExif(is, ExifInterface.Options.OPTION_ALL);

      ExifTag tag=exif.getTag(ExifInterface.TAG_ORIENTATION);
      int orientation=(tag==null ? -1 : tag.getValueAsInt(-1));

      if (orientation==8 || orientation==3 || orientation==6) {
        is=assets.open(ASSET_NAME);

        Bitmap original=BitmapFactory.decodeStream(is);
        Bitmap rotated=null;

        if (BuildConfig.ROTATE_BITMAP) {
          rotated=rotateViaMatrix(original, orientation);

          exif.setTagValue(ExifInterface.TAG_ORIENTATION, 1);
          exif.removeCompressedThumbnail();

          File output=
            new File(ctxt.getExternalFilesDir(null), "rotated.jpg");

          exif.writeExif(rotated, output.getAbsolutePath(), 100);

          MediaScannerConnection.scanFile(ctxt,
            new String[]{output.getAbsolutePath()}, null, null);
        }

        EventBus
          .getDefault()
          .postSticky(new ImageLoadedEvent(original, rotated, orientation));
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception processing image", e);
```

```
    }
  }
}
```

(from Camera/EXIFRotater/app/src/main/java/com/commonsware/android/exif/MainActivity.java)

We first get an `InputStream` on the particular image from `assets/` that we are to show (hard-coded as the `ASSET_NAME` constant).

We then create an `ExifInterface`, using the richer implementation from the aforementioned artifact. This `ExifInterface` has a few versions of `readExif()`, including one that can take our `InputStream` as input. We can then get the orientation tag value via calls to `getTag()` (to get an `ExifTag` for `TAG_ORIENTATION`), then `getValueAsInt()`. The latter method retrieves an integer tag value, with a supplied default value if the tag exists but does not have an integer value.

However, it is also possible that the tag does not exist. In fact, many JPEG images will lack this header, implying that the image is already in the correct orientation. So, we use the ternary operator and the default value to `getValueAsInt()` to get either the actual orientation tag numeric value or –1 if, for any reason, we cannot get that value.

If the orientation is 3, 6, or 8, we will want to show the image. So, we use `BitmapFactory` to load the image, via `decodeStream()`. If `ROTATE_BITMAP` is `true`, we do five things:

1. We rotate the `Bitmap` itself using a `Matrix`, in the `rotateViaMatrix()` method:

```
static private Bitmap rotateViaMatrix(Bitmap original, int orientation) {
  Matrix matrix=new Matrix();

  matrix.setRotate(degreesForRotation(orientation));

  return(Bitmap.createBitmap(original, 0, 0, original.getWidth(),
    original.getHeight(), matrix, true));
}
```

(from Camera/EXIFRotater/app/src/main/java/com/commonsware/android/exif/MainActivity.java)

1. We set the orientation tag to normal (1), reflecting the fact that we have oriented the image properly.
2. We remove any thumbnail from our EXIF metadata read from the original image

3. We write the revised EXIF data and the rotated `Bitmap` to a file on external storage, so we have a JPEG showing the rotated results yet including all of the original EXIF tags (excluding the orientation tag and thumbnail)
4. We tell `MediaScannerConnection` to scan this newly-created file, so it shows up in file managers, both on-device and on-desktop

If `ROTATE_BITMAP` is `false`, we instead handle the rotation in our `onImageLoaded()` method that is called when the `ImageLoadedEvent` is posted:

```
@Subscribe(sticky=true, threadMode=ThreadMode.MAIN)
public void onImageLoaded(ImageLoadedEvent event) {
  original.setImageBitmap(event.original);

  if (BuildConfig.ROTATE_BITMAP) {
    oriented.setImageBitmap(event.rotated);
  }
  else {
    oriented.setImageBitmap(event.original);
    oriented.setRotation(degreesForRotation(event.orientation));
  }
}
```

(from Camera/EXIFRotater/app/src/main/java/com/commonsware/android/exif/MainActivity.java)

Rather than show the rotated image in the lower `ImageView`, we show the original image, then rotate the `ImageView`.

Which of these two approaches — rotate the `ImageView` or rotate the image — is appropriate for you depends on your app. If all you need to do is show the image properly to the user, rotating the `ImageView` should be less memory-intensive. If, on the other hand, you need to save the corrected image somewhere for later use, you will need to rotate the image itself to make that correction.

And Then, There Are the Bugs

Some devices have buggy firmware, where they do not rotate the image themselves *nor* set the orientation tag in the image. Instead, they just ignore the whole issue. For these devices, we have no way of distinguishing between "images that need to be rotated, but we do not know the orientation" and "images that are fine and do not need to be rotated".

Your best option is to let the user manually request that the image be rotated (e.g., action bar "rotate" item).

Scanning with ZXing

If your objective is to scan a barcode, it is *much* simpler for you to integrate Barcode Scanner into your app than to roll it yourself.

Barcode Scanner – one of the most popular Android apps of all time — can scan a wide range of 1D and 2D barcode types. They offer an integration library that you can add to your app to initiate a scan and get the results. The library will even lead the user to the Play Store to install Barcode Scanner if they do not already have the app.

One limitation is that while the ZXing team (the authors and maintainers of Barcode Scanner) make the integration library available, they only do so in source form.

That sample project — `Camera/ZXing` – has a UI dominated by a "Scan!" button. Clicking the button invokes a `doScan()` method in our sample activity:

This passes control to Barcode Scanner by means of the integration JAR and the `IntentIntegrator` class. `initiateScan()` will validate that Barcode Scanner is installed, then will start up the camera and scan for a barcode.

Once Barcode Scanner detects a barcode and decodes it, the activity invoked by `initiateScan()` finishes, and control returns to you in `onActivityResult()` (as the Barcode Scanner scanning activity was invoked via `startActivityForResult()`). There, you can once again use `IntentIntegrator` to find out details of the scan, notably the type of barcode and the encoded contents:

```
public void onActivityResult(int request, int result, Intent i) {
  IntentResult scan=IntentIntegrator.parseActivityResult(request,
                                                         result,
                                                         i);

  if (scan!=null) {
    format.setText(scan.getFormatName());
    contents.setText(scan.getContents());
  }
}
```

(from Camera/ZXing/app/src/main/java/com/commonsware/android/zxing/ZXingDemo.java)

To use `IntentIntegrator` and `IntentResult`, the sample project has two modules: the `app/` module for the app, and a `zxing/` module containing those two classes

(and a rump `AndroidManifest.xml` to make the build tools happy). The `app/` module depends upon the `zxing` module via a `implementation project(':zxing')` dependency directive.

Some notes:

- Barcode Scanner's scanning activity only works in landscape
- Even though you are not using the camera directly yourself, you should consider including the `<uses-feature>` element declaring that you need a camera, if your app cannot function without barcodes
- If you wish to add Barcode Scanner logic directly to your app, and avoid the dependency on the third-party APK, that is possible, but the process for doing it is not well documented or supported

Videos: Letting the Camera App Do It

Just as `ACTION_IMAGE_CAPTURE` can be used to have a third-party app supply you with still images, there is an `ACTION_VIDEO_CAPTURE` on `MediaStore` that can be used as an `Intent` action for asking a third-party app capture a video for you. As with `ACTION_IMAGE_CAPTURE`, you use `startActivityForResult()` with `ACTION_VIDEO_CAPTURE` to find out when the video has been recorded.

There are two extras of note for `ACTION_VIDEO_CAPTURE`:

- `MediaStore.EXTRA_OUTPUT`, which indicates where the video should be written, and
- `MediaStore.EXTRA_VIDEO_QUALITY`, which should be an integer, either `0` for low quality/low size videos or `1` for high quality

If you elect to skip `EXTRA_OUTPUT`, the video will be written to the default directory for videos on the device (typically a "Movies" directory in the root of external storage), and the `Uri` you receive on the `Intent` in `onActivityResult()` will point to this file.

The impacts of skipping `EXTRA_VIDEO_QUALITY` are undocumented.

The `Media/VideoRecordIntent` sample project is a near-clone of the `Camera/FileProvider` sample from earlier in this chapter. Instead of requesting a third-party app take a still image, though, this sample requests that a third-party app record a video:

```
package com.commonsware.android.videorecord;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.net.Uri;
import android.os.Build;
import android.os.Bundle;
import android.provider.MediaStore;
import android.support.v4.content.FileProvider;
import java.io.File;
import java.util.List;

public class MainActivity extends Activity {
  private static final String EXTRA_FILENAME=
    BuildConfig.APPLICATION_ID+".EXTRA_FILENAME";
  private static final String AUTHORITY=
    BuildConfig.APPLICATION_ID+".provider";
  private static final String VIDEOS="videos";
  private static final String FILENAME="sample.mp4";
  private static final int REQUEST_ID=1337;
  private File output=null;
  private Uri outputUri=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (savedInstanceState==null) {
      output=new File(new File(getFilesDir(), VIDEOS), FILENAME);

      if (output.exists()) {
        output.delete();
      }
      else {
        output.getParentFile().mkdirs();
      }
    }
    else {
      output=(File)savedInstanceState.getSerializable(EXTRA_FILENAME);
    }

    outputUri=FileProvider.getUriForFile(this, AUTHORITY, output);

    if (savedInstanceState==null) {
      Intent i=new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
```

```
      i.putExtra(MediaStore.EXTRA_OUTPUT, outputUri);
      i.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);

      if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
        i.addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION |
          Intent.FLAG_GRANT_READ_URI_PERMISSION);
      }
      else {
        List<ResolveInfo> resInfoList=
          getPackageManager()
            .queryIntentActivities(i, PackageManager.MATCH_DEFAULT_ONLY);

        for (ResolveInfo resolveInfo : resInfoList) {
          String packageName = resolveInfo.activityInfo.packageName;

          grantUriPermission(packageName, outputUri,
            Intent.FLAG_GRANT_WRITE_URI_PERMISSION |
              Intent.FLAG_GRANT_READ_URI_PERMISSION);
        }
      }

      startActivityForResult(i, REQUEST_ID);
    }
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putSerializable(EXTRA_FILENAME, output);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==REQUEST_ID && resultCode==RESULT_OK) {
      Intent view=
          new Intent(Intent.ACTION_VIEW)
            .setDataAndType(outputUri, "video/mp4")
            .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

      startActivity(view);
      finish();
    }
  }
}
```

(from Media/VideoRecordIntent/app/src/main/java/com/commonsware/android/videorecord/MainActivity.java)

`onCreate()` of `MainActivity` starts by setting up a `File` object pointing to a `sample.mp4` file in internal storage. If the file already exists, `onCreate()` deletes it; otherwise it ensures that the directory already exists. We then go through much of the same headache that we did in the `ACTION_IMAGE_CAPTURE` scenario, creating a `Uri` for our `FileProvider` that points to our designated `File`, ensures that the video-recording app has read/write access to our `Uri`, before finally calling `startActivityForResult()`.

The call to `startActivityForResult()` will trigger the third-party app to record the video. When control returns to `MainActivity`, `onActivityResult()` creates an `ACTION_VIEW` `Intent` for the same `Uri`, then calls `startActivity()` to request that some app play back the video.

And, as before, we hold onto the `File` object via the saved instance state `Bundle`, and we only record the video if there is no such saved instance state `Bundle`, in case there is a configuration change causing our activity to be destroyed and recreated.

There is only one problem: this app is less likely to work on your device that did the `ACTION_IMAGE_CAPTURE` sample.

Camera apps need to be able to support `content` `Uri` values for `EXTRA_OUTPUT` for both still images and video. However, Google did not support this in their own camera app, decreasing the likelihood that anyone else supports it.

You can elect to use a `file` `Uri`, pointing to a location on external storage. However, that will require you to keep your `targetSdkVersion` at 23 or lower, as once you go above that, `file` `Uri` values are banned in `Intent` objects on Android 7.0.

Using a Camera Library

Relying upon third-party applications for taking pictures does introduce some challenges:

- Not all camera apps are created equal. Some implement `ACTION_IMAGE_CAPTURE` and `ACTION_VIDEO_CAPTURE` well... and others do not. Some might only ever give you a thumbnail, or some might not support all valid `Uri` values for writing out the output, and so on.
- Even within "valid" output, there can be variances. One common variation is how portrait images are handled. Some camera apps will write out an image that is actually in portrait mode. Some camera apps will write out an image

that is set up for landscape, but with an "EXIF header" in the JPEG data that tells image viewers to rotate the image to portrait. Unfortunately, not everything honors those headers, such as Android's own `BitmapFactory`.
- If the camera app uses a lot of system RAM, your app may be kicked out of RAM while the user is taking a picture. This should not be a problem, as your app's process is eligible to be terminated at any point when you are not in the foreground. However, it is a bit unexpected to think that taking a picture may cause you to have to switch to a fresh process.

The alternative to relying upon a third-party app is to implement camera functionality within your own app. For that, you have three major options:

1. Use the `android.hardware.Camera` API, added to Android way back in API Level 1, but marked as deprecated in API Level 21
2. Use the `android.hardware.camera2` API, added to Android in API Level 21 as a replacement for `android.hardware.Camera`, but therefore is only useful on its own if your `minSdkVersion` is 21 or higher
3. Use some third-party library that wraps around one or both of those APIs

Using the native camera APIs is possible but difficult. However, there are a few libraries that simplify the process. Here, we will look at two of them: CameraKit-Android and Fotoapparat.

CameraKit-Android

Debuting in 2017, CameraKit-Android is an MIT-licensed library, making it convenient for both open source and proprietary development. It offers a fair bit of flexibility in a simple API. It supports both taking photos and recording videos. However:

- The library was handed off from previous maintainers to a new team, leaving many outstanding issues in limbo
- The documentation is incorrect for several features

The `Camera/CameraKit` sample project demonstrates using CameraKit-Android. It has a two-button UI in its `MainActivity`, where one button will trigger taking a picture, while the other button triggers recording a video.

Adding the Dependency

CameraKit-Android is published as the `com.wonderkiln:camerakit-core` artifact on JCenter. However:

- The library depends upon `appcompat-v7`, despite not using classes from there
- The library depends upon Google Play Services, though only for one particular method

Hence, the sample app uses Gradle to exclude those two transitive dependencies and instead depend upon `support-compat`, since the sample app it not using `appcompat-v7` itself:

```
dependencies {
    implementation 'com.github.clans:fab:1.6.4'
    implementation 'com.githang:com-phillipcalvin-iconbutton:1.0.1@aar'
    implementation 'com.android.support:support-compat:27.0.2'
    implementation('com.wonderkiln:camerakit-core:0.13.1') {
      exclude group: 'com.android.support', module: 'appcompat-v7'
      exclude group: 'com.google.android.gms', module: 'play-services-vision'
    }
}
```

(from Camera/CameraKit/app/build.gradle)

The other two dependencies are for:

- An implementation of a floating action button (FAB), profiled in the chapter on the Design Support Library
- An icon button library, for use by the `MainActivity`

Adding the Preview Display

The actual photo- or video-taking activity is `CameraActivity`. It uses a `camera.xml` layout resource, which in our case contains a custom `CameraView` from CameraKit-Android, along with a FAB:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
```

```
  <com.wonderkiln.camerakit.CameraView android:id="@+id/camera"
    xmlns:camerakit="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:adjustViewBounds="true"
    camerakit:ckFlash="on" />

  <com.github.clans.fab.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_marginBottom="@dimen/fab_margin"
    android:layout_marginEnd="@dimen/fab_margin"
    android:layout_marginRight="@dimen/fab_margin"
    android:src="@drawable/ic_camera_black_24dp"
    app:fab_colorNormal="@color/ready" />
</RelativeLayout>
```

(from Camera/CameraKit/app/src/main/res/layout/camera.xml)

CameraKit-Android puts most of its configuration on its `CameraView`, whether via custom attributes or via corresponding getter/setter methods on the Java class. In this case, we are using the `ckFlash` attribute, indicating that we want flash mode to be enabled.

Other available attributes include:

- `ckFacing`, to indicate if you want the rear-facing (default) or front-facing camera
- `ckFocus`, to indicate what focus mode you want (defaults to continuous focus)
- `ckJpegQuality`, to indicate the quality factor to use when saving a photo as JPEG (defaults to 100, indicating maximum quality/maximum size)
- `ckVideoQuality`, to indicate the resolution of the video to capture, or to choose the highest-available or lowest-available resolution (defaults to a maximum of 480p, or 640x480)
- `ckZoom`, indicating that we want a pinch-to-zoom implementation, so pinch gestures control the camera's zoom level

Permissions

If you do not request runtime permissions, CameraKit-Android will do that for you as part of displaying that `CameraView`. In the case of this sample app, `MainActivity` extends the `AbstractPermissionActivity` seen elsewhere in this book, and it requests three permissions:

- `CAMERA`, for using the camera
- `WRITE_EXTERNAL_STORAGE`, as this is where videos are recorded by default
- `RECORD_AUDIO`, needed for recording videos, as they also capture audio off of a microphone

Integrating with the Lifecycle

`MainActivity` invokes `CameraActivity` by means of two `static` methods on `CameraActivity`: `takePhoto()` and `recordVideo()`. These simply start up `CameraActivity` with `EXTRA_IS_PHOTO` indicating which mode the camera should use:

```
public static void takePhoto(Activity requester, int requestCode) {
  Intent i=new Intent(requester, CameraActivity.class)
    .putExtra(EXTRA_IS_PHOTO, true);

  requester.startActivityForResult(i, requestCode);
}

public static void recordVideo(Activity requester, int requestCode) {
  Intent i=new Intent(requester, CameraActivity.class)
    .putExtra(EXTRA_IS_PHOTO, false);

  requester.startActivityForResult(i, requestCode);
}
```

(from Camera/CameraKit/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

In `onCreate()`, we load up our layout and retrieve the `CameraView`, holding onto it in a `camera` field:

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.camera);

  isPhoto=getIntent().getBooleanExtra(EXTRA_IS_PHOTO, true);
```

```
    camera=findViewById(R.id.camera);
    fab=findViewById(R.id.fab);

    fab.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View view) {
        if (isPhoto) {
          takePhoto();
        }
        else {
          recordVideo();
        }
      }
    });

    if (!isPhoto) {
      fab.setImageResource(R.drawable.ic_videocam_black_24dp);
    }

    camera.addCameraKitListener(new CameraKitEventListenerAdapter() {
      @Override
      public void onImage(CameraKitImage image) {
        // TODO: do something with picture

        setResult(RESULT_OK);
        finish();
      }

      @Override
      public void onVideo(CameraKitVideo video) {
        // TODO: do something with video file

        setResult(RESULT_OK);
        finish();
      }
    });
  }
```

(from Camera/CameraKit/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

In addition, we:

- Set up the FAB with the proper icon and action, depending upon whether we are taking a picture or recording a video
- Add a listener to the `CameraView`, as will be discussed shortly

We need to forward `onPause()` and `onResume()` events from the hosting activity or fragment along to the `CameraView`, so the custom view knows when it should be releasing the camera:

Taking a Picture

Taking a picture with CameraKit-Android is merely a matter of calling `captureImage()`:

```
private void takePhoto() {
  camera.captureImage();
  fab.setEnabled(false);
}
```

(from Camera/CameraKit/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

When the picture is taken, the `CameraKitEventListenerAdapter` instance that we registered via `addCameraKitListener()` on the `CameraView` will be called with `onImage()`. We are handed a `CameraKitImage` object, which has `getBitmap()` and `getJpeg()` methods to give us the photo in memory. The latter is the original data off of the camera; the former returns a `Bitmap` created from that JPEG. Note that for a high-resolution image, you may not be able to create the `Bitmap` without running out of memory. In a real app, we would do something with the photo, such as saving it to disk on a background thread.

Recording a Video

To start recording a video, call `start()`; to stop recording a video, call `stopVideo()`:

```
private void recordVideo() {
  if (isRecording) {
    camera.stopVideo();
    finish();
  }
  else {
    fab.setColorNormalResId(R.color.recording);
    fab.setImageResource(R.drawable.ic_stop_black_24dp);
    isRecording=true;
    camera.start();
  }
}
```

(from Camera/CameraKit/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

When the video is taken, our listener will be called with `onVideo()`, where we are given a `CameraKitVideo` object. That has a `getFile()` method, which contains a `File` object pointing to wherever the library elected to save the video. By default, this will be in your app's `getExternalFilesDir()` location, so if your `minSdkVersion` is 19 or higher, you will not need the `WRITE_EXTERNAL_STORAGE` permission. A production app would do something with this file.

Fotoapparat

Also debuting in 2017, the Apache-licensed Fotoapparat has a similar flow to CameraKit-Android: add a custom `View` for the preview and call methods to take pictures. However:

- Fotoapparat puts most of the configuration on a `Fotoapparat` object
- The `Fotoapparat` API offers more a few more features than does CameraKit-Android's API, though with a somewhat funky fluent API
- As of July 2017, Fotoapparat only supports taking photos, not recording videos

The `Camera/Fotoapparat` sample project is a clone of the CameraKit-Android sample, with Fotoapparat as the camera library. Since Fotoapparat does not support recording videos, the code related to that was removed, and its button in the UI is disabled.

Adding the Dependency

Unlike CameraKit-Android, Fotoapparat is distributed through jitpack.io. That is not a pre-configured artifact repository, so we add it, via the `allprojects` closure in the project-level `build.gradle` file:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    repositories {
        jcenter()
        google()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}
```

```
allprojects {
    repositories {
        jcenter()
        google()
        maven { url 'https://jitpack.io' }
    }
}
```

(from Camera/Fotoapparat/build.gradle)

However, similarly to CameraKit-Android, Fotoapparat unnecessarily depends upon `appcompat-v7`, so once again we filter it out:

```
dependencies {
    implementation 'com.github.clans:fab:1.6.4'
    implementation 'com.githang:com-phillipcalvin-iconbutton:1.0.1@aar'
    implementation 'com.android.support:support-compat:26.1.0'
    implementation 'io.fotoapparat.fotoapparat:library:1.3.0'
}

android {
```

(from Camera/Fotoapparat/app/build.gradle)

Permissions

Fotoapparat will merge in the `<uses-permission>` element for the `CAMERA` permission to your manifest, but it does not attempt to request that permission at runtime. Apps using Fotoapparat have to handle that themselves, before attempting to use the camera.

The sample app uses the same `AbstractPermissionsActivity` as before, albeit only with the `CAMERA` permission, since that is the only one needed here.

Adding the Preview Display

Fotoapparat has its own `CameraView`. However, whereas CameraKit-Android puts its configuration on the `CameraView`, Fotoapparat does not. As a result, other than sizing and positioning the `CameraView` where you want it, little else is needed:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
```

```
  <io.fotoapparat.view.CameraView android:id="@+id/camera"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

  <com.github.clans.fab.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_marginBottom="@dimen/fab_margin"
    android:layout_marginEnd="@dimen/fab_margin"
    android:layout_marginRight="@dimen/fab_margin"
    android:src="@drawable/ic_camera_black_24dp"
    app:fab_colorNormal="@color/ready" />
</RelativeLayout>
```

(from Camera/Fotoapparat/app/src/main/res/layout/camera.xml)

Configuring the Fotoapparat

However, what we remove from the layout goes into Java code. Specifically, Fotoapparat has a `Fotoapparat` class. We need to build an instance of a `Fotoapparat` and connect it to the `CameraView`.

`CameraActivity` handles this in `onCreate()`:

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.camera);

  camera=findViewById(R.id.camera);
  fab=findViewById(R.id.fab);

  fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
      takePhoto();
    }
  });

  fotoapparat=Fotoapparat
    .with(this)
    .into(camera)
```

```
      .previewScaleType(ScaleType.CENTER_CROP)
      .photoSize(biggestSize())
      .lensPosition(back())
      .focusMode(firstAvailable(continuousFocus(), autoFocus(), fixed()))
      .flash(firstAvailable(autoRedEye(), autoFlash()))
      .build();
  }
```

(from Camera/Fotoapparat/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

The `Fotoapparat` API is designed for significant use of statically imported methods. This is akin to how libraries like JUnit and Espresso work, reducing the visual clutter of the main Java code at the cost of a bunch of additional `import` statements.

Fotoapparat is also significantly more configurable than is CameraKit-Android. There are many more options, and more complex options, for controlling how the camera is used.

So, in `onCreate()`, we:

- Create a `FotoapparatBuilder` via `with()`, which takes a `Context`
- Tie it to the `CameraView` via `into()`
- Indicate that we want the camera preview to fill the available space, at the cost of cropping portions that do not match our `CameraView` aspect ratio, via `previewScaleType(ScaleType.CENTER_CROP)`
- Indicate that we want to take the highest resolution photo possible, via `photoSize(biggestSize())`
- Indicate that we want the rear-facing camera via `lensPosition(back())`
- Request either continuous focus, auto-focus, or fixed focus, whichever one is available, via `focusMode(firstAvailable(continuousFocus(), autoFocus(), fixed()))`
- Request either auto-redeye or auto flash, whichever one is available, via `flash(firstAvailable(autoRedEye(), autoFlash()))`

Then, we `build()` the `Fotoapparat` from the `FotoapparatBuilder`.

Methods like `photoSize()`, `focusMode()`, and `flashMode()` take a "selector" object. Sometimes, that might be a specific setting (e.g., `biggestSize()` for `photoSize()`). Sometimes, that might be more of a "functor" object, one that makes a decision of other selectors to use based upon availability (e.g., `firstAvailable()`). This leads to a very expressive API, at the cost of a fairly convoluted set of objects and methods — writing an API like this can get complicated rather quickly.

Integrating with the Lifecycle

As with CameraKit-Android, we need to forward lifecycle events along to the `Fotoapparat` object. In this case, though, we need to forward `onStart()` and `onStop()`, not `onPause()` and `onResume()`:

```
@Override
protected void onStart() {
  super.onStart();
  fotoapparat.start();
}

@Override
protected void onStop() {
  fotoapparat.stop();
  super.onStop();
}
```

(from Camera/Fotoapparat/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

Taking a Picture

The `Fotoapparat` object has a `takePicture()` method which, as you might expect, takes a picture. It gives you back a `PhotoResult` object, which you can use to work with the resulting photo. In the sample app, we call `toBitmap()` on it, then use `whenAvailable()` to get a callback when the image is ready. This gives us a `BitmapPhoto` object, from which we can get the JPEG bytes, akin to the CameraKit-Android example:

```
private void takePhoto() {
  fab.setEnabled(false);

  PhotoResult result=fotoapparat.takePicture();

  result.toBitmap().whenAvailable(new PendingResult.Callback<BitmapPhoto>() {
    @Override
    public void onResult(BitmapPhoto bitmapPhoto) {
      // TODO: do something with picture

      setResult(RESULT_OK);
      finish();
    }
  });
}
```

(from Camera/Fotoapparat/app/src/main/java/com/commonsware/android/camera/CameraActivity.java)

If, however, your objective is to save the photo to a file, simply call `saveToFile()` on the `PhotoResult` object. This will asynchronously write your JPEG to the supplied `File` location, and it returns a `PendingResult`, as `toBitmap()` does, for you to attach a callback to find out when the save operation is complete.

Directly Working with the Camera

Of course, you can bypass these third-party apps and libraries, electing instead to work directly with the camera if you so choose. This is very painful, as will be illustrated in the next chapter.

Working Directly with the Camera

Letting third-party apps take the pictures and videos for you is all well and good, but there will be times where you need more control than that. It is possible for you to work directly with the device cameras. However, doing is exceptionally complicated.

Part of that complexity is because Android presently has three separate APIs for working with the camera:

- `android.hardware.Camera` for taking still photos
- `android.hardware.camera2` for taking still photos on Android 5.0+ devices
- `MediaRecorder` for recording videos

This chapter will attempt to outline the basic steps for using these APIs.

Prerequisites

This chapter assumes that you have read the previous chapter covering `Intent`-based uses of the camera and the chapter on audio recording.

Notes About the Code Snippets

The code snippets shown in this chapter are here purely to illustrate how to call certain APIs. They are not from any particular sample project, as a sample project small enough to fit in a book would be riddled with bugs and limitations.

A Tale of Two APIs

As noted in the introduction to this chapter, there are three APIs for working with the camera. One — `MediaRecorder` — is focused purely on recording videos. It relies on you using one of the other two APIs for setting up the camera preview, so the user can see what will be recorded. Those other two APIs exist for taking still photos, where one (`android.hardware.camera2`) is substantially newer.

android.hardware.Camera

The original camera API is based around the `android.hardware.Camera` class.

(NOTE: there is another `Camera` class, in `android.graphics`, that is not directly related to taking pictures)

Instances of this class represent an open camera, where you call methods on the `Camera` to do things like take pictures. You also work extensively with a `Camera.Parameters` object, where you can determine a number of key characteristics about the camera (e.g., what the available resolutions are for pictures) and set up the particular results that you want.

This API works on all Android devices.

android.hardware.camera2

The original camera API worked, albeit with some difficulty. However, it was fairly limited, as it was designed primarily around the smartphone camera capabilities of 2005-2010. Nowadays, device manufacturers have access to much more powerful camera modules from chipset manufacturers like Qualcomm. Android needed a more powerful API to accommodate the current hardware, and a more flexible API to be able to adjust to changes over time.

Hence, Android 5.0 brought a new API, based on a series of classes in the `android.hardware.camera2` package. On the plus side, these offer much greater capability. They are also designed with asynchronous work in mind, off-loading slow or complex operations onto background threads for you. However, on the whole, the API is more complicated, much less documented, and substantially different than the original API.

It is also only available on Android 5.0 devices. If your `minSdkVersion` is 21 or higher, that is not a problem. If, however, you are aiming to support older devices than that, you have two choices:

1. Stick with the original API for all devices
2. Use the original API for older devices and the newer API for newer devices

The latter might allow you to offer more features to users of those newer devices, but it does roughly double the work required to implement camera logic in your app.

MediaRecorder

`MediaRecorder` is responsible for both audio recording and video recording. `MediaRecorder` has a fairly limited API, one that has not changed substantially since 2011. However, if you use it carefully, it works. It works in tandem with either camera API — you use the camera APIs to show the user what will be recorded, and you use `MediaRecorder` to actually do the recording.

However, `MediaRecorder` has a number of issues, such as a fair bit of delay between when you ask it to begin recording and when it actually *does* begin recording. This makes it a poor choice for fast-twitch video recording purposes. Some apps, notably Vine, have elected to skip using `MediaRecorder`. Instead, they use the regular camera APIs. These APIs, among other things, give you access to the preview frames that are used to show the user what is visible through the camera lens. With a fair amount of work, you can stitch those together into a video. Needless to say, this is a beyond-advanced topic that is well outside the scope of this book.

The APIs That You (Probably) Can't Use

The aforementioned APIs are all part of the Android SDK. For camera apps that ship with devices, those apps are not limited to these APIs. Device manufacturers are welcome to create apps that use internal proprietary APIs for their devices.

Hence, when it comes to determining what is and is not possible through the camera APIs, it is important to compare to other third-party camera apps, more so than manufacturer-supplied apps. Manufacturers can "cheat"; you cannot.

Performing Basic Camera Operations

Cameras have some key functionality:

- Showing a preview to the user, so the user can see in real time what the camera lens sees, so the user can frame a picture
- Take a still picture
- Record a video

In the following sections, we will outline what is required to perform these operations using the various APIs.

Permissions

First, you need permission to use the camera. That way, when end users install your application, they will be notified that you intend to use the camera, so they can determine if they deem that appropriate for your application.

You simply need the `CAMERA` permission in your `AndroidManifest.xml` file, along with whatever other permissions your application logic might require.

If you plan to record video, using `MediaRecorder`, you will also want to request the `RECORD_AUDIO` permission.

And, if you were planning on storing pictures or videos out on external storage, you probably need the `WRITE_EXTERNAL_STORAGE` permission. The exception would be if your `minSdkVersion` is 19 or higher and you are only storing those files in locations that are automatically read/write for your app, such as `getExternalFilesDir()` or `getExternalCacheDir()`.

Note that all three of these permissions (`CAMERA`, `RECORD_AUDIO`, and `WRITE_EXTERNAL_STORAGE`) are part of the Android 6.0 runtime permission system. If your app has a `targetSdkVersion` of 23 or higher, you will need to request those permissions at runtime. If your app has a lower `targetSdkVersion`, while you will not have to do anything special for your app, bear in mind that the user can still revoke your access to those capabilities, and so you may find lots of devices that claim to support a camera but just do not seem to have any cameras available when you try to use one.

Features

Your manifest also should contain one or more `<uses-feature>` elements, declaring what you need in terms of camera hardware. By default, asking for the `CAMERA` permission indicates that you *need* a camera. More specifically, asking for the `CAMERA` permission indicates that you need an *auto-focus* camera.

The following sections outline some common scenarios and how to handle them.

A Camera is Optional

If you would like a camera, but having one is not essential for the use of your app, put the following `<uses-feature>` element in your manifest:

```
<uses-feature android:name="android.hardware.camera" android:required="false" />
```

This indicates that you would like a camera, but it is not required. This reverses the default established by the `CAMERA` permission.

A Camera is Required

Technically, you would not need any `<uses-feature>` element in your manifest to indicate that you need a camera, as the `CAMERA` permission would handle that for you. However, it is good form to explicitly declare it anyway:

```
<uses-feature android:name="android.hardware.camera" android:required="true" />
```

Not only does that make your manifest more self-documenting, but it also helps protect you in case the default behavior of the `CAMERA` permission changes.

Other Camera Features

There are three other camera features that you could consider having `<uses-feature>` elements for:

1. `android.hardware.camera.autofocus`, to indicate whether or not the device needs a camera with auto-focus capability.
2. `android.hardware.camera.flash`, to indicate whether or not the device must support a camera flash
3. `android.hardware.camera.front`, to indicate whether or not the app needs a front-facing camera specifically (`android.hardware.camera` requests a rear-facing camera)

Of these, the only one you should definitely include in your app is `android.hardware.camera.autofocus`, once again because of the default effects of requesting the `CAMERA` permission. In particular, if you do not absolutely need auto-focus capabilities, you can use `android:required="false"` to reverse the `CAMERA` default requirement.

Finding Out What Cameras Exist

Some devices will have just a rear-facing camera. Some will have just a front-facing camera. Some will have both cameras. Some will have no cameras. And, in theory at least, some could have yet more camera options.

At some point, you are likely to need to find out what cameras exist on the device that you are running on. Perhaps you need a particular camera (e.g., a front-facing camera for your "selfie"-focused app). Or, perhaps you want to allow your users to switch between cameras on the fly.

android.hardware.Camera

The simplest way to choose a camera is to not choose at all, and arrange to open the default camera. That default camera is the first rear-facing camera on the device. However, devices that have *no* rear-facing cameras effectively have no default camera, and so going with the default is rarely the correct choice. Instead, you should iterate over the available cameras, to find the one that you want.

To find out how many cameras there are for the current device, you can call the static `getNumberOfCameras()` method on the `Camera` class.

To find out details about a particular camera, you can call the static `getCameraInfo()` method on `Camera`. This takes two parameters:

- the ID of the camera to open, which will be a number from 0 to the number of available camera minus 1
- a `Camera.CameraInfo` object, into which `getCameraInfo()` will pour details about the camera

The most notable field on `Camera.CameraInfo` is `facing`, which tells you if this is a rear-facing (`Camera.CameraInfo.CAMERA_FACING_BACK`) or front-facing (`Camera.CameraInfo.CAMERA_FACING_FRONT`) camera.

For example, the following code snippet could be used to identify the first front-facing camera:

```
int chosen=-1;
int count=Camera.getNumberOfCameras();
Camera.CameraInfo info=new Camera.CameraInfo();
```

```
for (int cameraId=0; cameraId < count; cameraId++) {
  Camera.getCameraInfo(cameraId, info);

  if (info.facing==Camera.CameraInfo.CAMERA_FACING_FRONT) {
    chosen=cameraId;
    break;
  }
}
```

If `chosen` remains at a value of `-1`, you know that there is no front-facing camera available to you, and you would need to decide how you wish to proceed, if you really wanted such a camera.

android.hardware.camera2

With the original camera API, your main entry point is the `Camera` class. With the Android 5.0+ camera API, your main entry point is a `CameraManager`. This is another system service, one you can retrieve by calling `getSystemService()` on a `Context`, asking for the `CAMERA_SERVICE`:

```
CameraManager mgr=
    (CameraManager)ctxt.
        getApplicationContext().
        getSystemService(Context.CAMERA_SERVICE);
```

You will notice here that we are specifically calling `getSystemService()` on the `Application` context. That is because there is a bug in Android 5.0 where `CameraManager` leaks the `Context` that creates it. This bug has been fixed in Android 5.1. However, to be safe, you are better off retrieving this system service via the singleton `Application` object, as there is no risk of a memory leak (singletons are "pre-leaked", as it were).

Given a `CameraManager`, you can call `getCameraIdList()` to get a list of camera IDs. These are strings, not integers as they were with the original camera API.

To learn more about the camera, you can ask the `CameraManager` to give you a `CameraCharacteristics` object for a given camera ID. The `CameraCharacteristics` object has all sorts of information about the camera, including what direction it is facing. `CameraCharacteristics` behaves a lot like a `HashMap`, in that you use `get()` and a key to retrieve a value, such as `CameraCharacteristics.LENS_FACING` to determine the camera's facing direction.

So, the code snippet for the first front-facing camera using a `CameraManager` named `mgr`, would be something like:

```
String chosen=null;

for (String cameraId : mgr.getCameraIdList()) {
  CameraCharacteristics cc=mgr.getCameraCharacteristics(cameraId);

  if
(cc.get(CameraCharacteristics.LENS_FACING)==CameraCharacteristics.LENS_FACING_FRONT) {
    chosen=cameraId;
    break;
  }
}
```

Here, a value of `null` would indicate that there is no available front-facing camera.

Opening and Closing a Camera

Once you decide which camera you wish to use, you will eventually need to "open" it. This gives your app access to that camera, and blocks other app's access while you have it open. You need to open a camera before you can use that camera to take pictures, record video, etc.

Eventually, when you are done with the camera, you should close it, to allow other apps to have access to the camera again. If you fail to close it, until your process is terminated, the camera is inaccessible.

android.hardware.Camera

Old code samples would open the camera by calling a zero-parameter static `open()` method on the `Camera` class. This opens the default camera, and as noted above, this is rarely a good idea. However, it is your only option on API Level 8 and below, if you are still supporting such devices, as those devices only supported a single camera.

Instead, if you have the ID of the camera that you wish to open, call the one-parameter static `open()` method, passing in the ID of the camera.

Both flavors of `open()` return an instance of `Camera`, which you can hold onto in your activity or fragment that is working with the camera.

While you have access to this camera, no other process can. Hence, it is important to release the camera when you are no longer needing it. To release the camera, call `release()` on your `Camera` instance, after which it is no longer safe to use the camera. A common pattern is to `open()` the camera in `onStart()` or `onResume()` and `release()` it in `onPause()` or `onStop()`, so you tie up the camera only while you are in the foreground.

android.hardware.camera2

Opening and closing a camera is a *lot* more complicated with the Android 5.0+ camera API.

Partly, that complexity seems to be due to a threading limitation with `CameraManager` — while we want to do long tasks related to the camera on background threads, `CameraManager` itself is not free-threaded when it comes to opening and closing cameras. Hence, we need to use some form of thread synchronization to make sure that we are not trying to open and close cameras simultaneously.

Partly, that complexity is that the way that `CameraManager` deals with background operations is via a `Handler` tied to a `HandlerThread`. `HandlerThread`, as the name suggests, is a `Thread` which has all the associated bits to support a `Handler`. The main application thread itself is a `HandlerThread` (or, close enough), but we specifically want to use a background thread, so we do not tie up the main application thread. So, we need to create and manage our own `HandlerThread` and `Handler`.

So, the first thing you will need to do is set up a `HandlerThread`, such as in a data member of some class:

```
final private HandlerThread handlerThread=new HandlerThread(NAME,
    android.os.Process.THREAD_PRIORITY_BACKGROUND);
```

Here, `NAME` is some string to identify this thread (used in places like the list of running threads in DDMS). The second parameter is the thread priority; in general, you want your own `HandlerThread` instances to have background priority.

Creating the `HandlerThread` instance does not actually start the thread, any more than creating a `Thread` object starts the thread. Instead, you need to call `start()` when you want the thread to begin working its message loop. Any time after this

point, it is safe to create a `Handler` for that `HandlerThread`, by getting the `Looper` from the `HandlerThread` and passing it to the `Handler` constructor:

```
handlerThread.start();
handler=new Handler(handlerThread.getLooper());
```

(You might wonder why a class named `HandlerThread`, designed to work with a `Handler`, lacks any methods to give you such a `Handler`. Lots of people wonder this, so you are not alone.)

Next, to actually open the camera, you will need to call `openCamera()` on your `CameraManager`, supplying:

- the ID of the camera that you wish to open
- a `CameraDevice.StateCallback` instance
- the `Handler` that you created for your `HandlerThread`

But, we want to make sure that we are not trying to open or close another camera while all of this is going on, so we need to use some sort of Java thread synchronization for that, such as a `Semaphore`:

```
final private Semaphore lock=new Semaphore(1);
```

Then, we can consider opening the camera, once we obtain the lock:

```
if (!lock.tryAcquire(2500, TimeUnit.MILLISECONDS)) {
  throw new RuntimeException("Time out waiting to lock camera opening.");
}

mgr.openCamera(cameraId, new DeviceCallback(), handler);
```

You will notice that we do not release the lock here, as we need to keep the lock until the camera has completed opening.

`CameraDevice.StateCallback` is an `abstract` class, so we usually have to create some dedicated subclass for it. There are three `abstract` methods that we will need to implement: `onOpened()`, `onError()`, and `onDisconnected()`. Plus, we will typically want to implement `onClosed()`, even though there is a default implementation of this callback.

`onOpened()` will be called when the camera is open and is ours to use. We are passed a `CameraDevice` object representing our open camera, and it is our job to hold onto

this device while we have the camera open. The big thing that we need to do in `onOpened()` is release that `lock` that we obtained when we tried opening the camera. This is also a fine time to consider starting to show camera previews to the user, and we will see how to do that in upcoming sections of the book.

`onError()` will be called if there is some serious error when trying to open or use the camera. We are passed an error code to indicate what sort of problem we encountered. It could be that the camera is already in use (`ERROR_CAMERA_IN_USE`), or that while the camera exists, we do not have access to it due to device policy (`ERROR_CAMERA_DISABLED`), or that there was a general problem with this specific camera (`ERROR_CAMERA_DEVICE`) or with the overall camera engine (`ERROR_CAMERA_SERVICE`).

`onDisconnected()` will be called if we no longer can use the camera, for reasons other than our closing it ourselves. We are supposed to close the `CameraDevice`, if we have one, as the camera is no longer usable.

To close the camera, whether in response to `onDisconnected()` or because you are simply done with the camera, call `close()` on the `CameraDevice`, inside of the `lock`:

```
try {
  lock.acquire();
  cameraDevice.close();
  cameraDevice=null;
}
finally {
  lock.release();
}
```

Note that `close()` is a synchronous call, and so we can `release()` our `lock` in a `finally` block.

Our `CameraDevice.StateCallback` will be called with `onClosed()`, to let us know that the close operation has completed.

Setting Up a Preview Surface

The camera preview is basically a stream of images, taken by the camera, usually at less than full resolution. Mostly, that stream is to be presented to the user on the screen, to help them “see what the camera sees”, so they can line up the right picture.

For presenting the preview stream to the user, there are two typical solutions: `SurfaceView` and `TextureView`.

SurfaceView for the Camera

`SurfaceView` is used as a raw canvas for displaying all sorts of graphics outside of the realm of your ordinary widgets. In this case, Android knows how to display a live look at what the camera sees on a `SurfaceView`, to serve as a preview pane. A `SurfaceView` is also used for video playback, and a variation of `SurfaceView` called `GLSurfaceView` is used for OpenGL animations.

That being said, `SurfaceView` is a subclass of `View`, and so it can be added to your UI the same as any other widget:

- Include it in a layout
- Return it as the `View` from `onCreateView()` of a `Fragment`
- Instantiate it in Java and add it to some container via `addView()`
- Etc.

If your app will support API Level 10 and older, you will want to call `getSurfaceHolder().getType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS)` on the `SurfaceView`. A "push buffers" `SurfaceView` is one designed to have images pushed to the surface, usually from video playback or camera previews. A `SurfaceHolder` is a quasi-controller object for the `SurfaceView` — most interactions with the `SurfaceView` come by way of the `SurfaceHolder`. This bit of configuration is not needed on API Level 11 and higher, as Android handles it for us automatically as the `SurfaceView` is put to use.

TextureView for the Camera

`SurfaceView`, however, has some limitations. This is mostly tied back to the way it works, by "punching a hole" in the UI to allow some lower-level component (like the camera) to render stuff into it. While there is a transparent layer on top of this "hole", for use in alpha-compositing in any overlapping widgets, the `SurfaceView` *content* is not rendered as part of the normal view hierarchy. The net effect is that you cannot readily move, animate, or otherwise transform a `SurfaceView`.

`TextureView` was added in API Level 14 and works for camera previews as of API Level 15. `TextureView` serves much the same role as does `SurfaceView`, for showing camera previews, playing videos, or rendering OpenGL scenes. However,

`TextureView` behaves as a regular `View` and so therefore can be animated and such without issue.

However, the cost is in performance. `TextureView` relies upon the GPU to do more work, and therefore `TextureView` is a bit less performant than is a `SurfaceView`. Most camera apps will not show a difference.

Showing the Previews

To show previews, you need to create your surface (`SurfaceView` or `TextureView`) and have it be part of your UI. Then, you can teach your opened camera to show previews on that surface.

android.hardware.Camera

The biggest thing that we need to do in the original camera API to configure the preview is determine what size of preview images should be used. Devices cannot support arbitrary-sized previews. Instead, we need to ask the camera what preview sizes it supports, choose one, then configure the camera to use that specific preview size.

To do any of this, we need the `Camera.Parameters` associated with our chosen and open `Camera`. `Camera.Parameters` serves two roles:

- It tells us what is possible, in terms of camera capabilities, above and beyond the limited information reported by `Camera.Info`
- It is where we stipulate what behavior we want, by updating the parameters and associating the updated parameters with the `Camera`

Getting the `Camera.Parameters` object from a `Camera` is a simple matter of calling `getParameters()`.

To find out what the valid preview sizes are, we can call `getSupportedPreviewSizes()` on the `Camera.Parameters` object. This will return a `List` of `Camera.Size` objects, with each `Camera.Size` holding a `width` and a `height` as integers.

Choosing a preview size is a bit of an art form. Too big of a preview size is wasteful from a performance standpoint. Too small of a preview size results in a grainy preview. And, as will be seen later in this chapter, the difference in aspect ratio between your surface and your preview size will need to be taken into account. We

will explore choosing preview sizes a bit more later in this chapter. For the moment, assume that we have sifted through the available preview sizes and have chosen something suitable. Whatever size you choose, you can pass to `setPreviewSize()` on the `Camera.Parameters`.

Then, you can call `setParameters()` on the `Camera`, passing in your modified `Camera.Parameters` object, to affect this change.

You will wind up with a block of code resembling:

```
Camera.Parameters parameters=camera.getParameters();
Camera.Size previewSize=chooseSomePreviewSize(parameters.getSupportedPreviewSizes());

parameters.setPreviewSize(previewSize.width, previewSize.height);

camera.setParameters(parameters);
```

(where `chooseSomePreviewSize()` is a method of your own design)

Given that, in principle, there are just three more steps:

1. Attach your preview surface to the `Camera` by calling `setPreviewDisplay()` (if you are using a `SurfaceView`) or `setPreviewTexture()` (if you are using a `SurfaceTexture`)
2. Show the preview on-screen by calling `startPreview()` on the `Camera`
3. Stop showing the preview by calling `stopPreview()` on the `Camera`

However, timing is important.

You also cannot call `setPreviewDisplay()` or `startPreview()` before your preview surface is ready. To know when *that* is, you will need to register a listener with your surface:

- You can register a `SurfaceHolder.Callback` with the `SurfaceHolder` of your `SurfaceView` by calling `addCallback()` on the `SurfaceHolder`. Your `SurfaceHolder.Callback` will be called with `surfaceChanged()` when the surface is ready for use, at which point it is safe to call `setPreviewDisplay()` and `startPreview()`.
- You can register a `TextureView.SurfaceTextureListener` with your `TextureView` by means of the `setSurfaceTextureListener()` call. Your `TextureView.SurfaceTextureListener` will be called with

`onSurfaceTextureAvailable()` at the point in time when it is safe to call `setPreviewTexture()` and `startPreview()`.

You also need to stop the preview before you `release()` the `Camera`. And, as we will see later in this chapter, you also need to restart your preview after taking a photo.

android.hardware.camera2

Once the camera is opened — even right from within the `onOpened()` method of your `CameraDevice.StateCallback` — you can request to have preview frames be pushed to your desired preview surface.

First, strangely enough, you are going to need to choose the resolution of the picture that you wish to take. You might think that this would be delayed until a later point, such as when we actually go to take a picture, but the API seems to want it right away.

To find out the possible resolutions, you need to request a `StreamConfigurationMap` from the `CameraCharacteristics`:

```
CameraCharacteristics cc=mgr.getCameraCharacteristics(cameraId);
StreamConfigurationMap map=
  cc.get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP);
```

(where `cameraId` is the ID of the camera that you are working with)

From there, you can get an array of `Size` objects via a call to `getOutputSizes()`. Curiously, `getOutputSizes()` takes a Java class object, identifying the use case for the frames to be generated by the camera. So, passing `SurfaceTexture.class` would give you preview frame resolutions, but passing `ImageFormat.JPEG` would give you picture resolutions (at least, for images to be encoded in JPEG format).

So, you can get your roster of available picture sizes via:

```
CameraCharacteristics cc=mgr.getCameraCharacteristics(cameraId);
StreamConfigurationMap map=
  cc.get(CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP);
Size[] rawSizes=map.getOutputSizes(ImageFormat.JPEG);
```

From there, you will need to choose a size. This process can be a bit interesting; some notes about it appear later in this chapter. But, for example, you might choose

the size that is the highest resolution, as determined by the total area (width times height).

Next, you are going to need to set up an `ImageReader`. Typically this is done via the `newInstance()` factory method, which takes four parameters:

- The width and height of the desired resolution of the *picture* that you wish to later take with the camera
- The image format to use (e.g., `ImageFormat.JPEG`) for those pictures
- How many simultaneous frames will be needed (typical value: 2)

```
ImageReader reader=ImageReader.newInstance(pictureSize.getWidth(),
          pictureSize.getHeight(), pictureFormat, 2);
```

Then, you need a `Surface` associated with your preview surface. For example, you can call `getSurfaceTexture()` on a `TextureView` to get a `SurfaceTexture`, then pass it to the `Surface` constructor to get the associated `Surface` object.

Next, you can call `createCaptureSession()` on the `CameraDevice` representing the opened camera. This takes three parameters:

- An `ArrayList` of `Surface` objects, for every places that the camera driver needs to route frames towards. Typically, you will have two elements in this list: the `Surface` for your preview surface and the `Surface` that you get from your `ImageReader` by calling `getSurface()` on it.
- A `CameraCaptureSession.StateCallback` instance, to be notified about state changes in the frame-capturing process
- The `Handler` tied to your `HandlerThread`

```
cameraDevice
  .createCaptureSession(Arrays.asList(surface, reader.getSurface()),
                        new PreviewCaptureSession(), handler);
```

(where `PreviewCaptureSession` is some subclass of `CameraCaptureSession.StateCallback`)

That actually does not begin the previews. Instead, it configures the camera to indicate that it is possible to do previews.

To continue the work for getting the previews rolling, in the `onConfigured()` callback method on your `CameraCaptureSession.StateCallback`, you can create a `CaptureRequest.Builder` that you can use for configuring the camera to capture

preview frames. You get one of those by calling `createCaptureRequest()` on the `CameraDevice`, passing in an `int` indicating the general type of request that you are creating, such as `TEMPLATE_PREVIEW` for preview frames:

```
CaptureRequest.Builder b=
  cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
```

You then call `setTarget()` on the `Builder`, supplying the `Surface` onto which the captured frames will be written. For previews, that target is the `Surface` associated with your preview surface.

You can also call `set()` on the `Builder` to configure various options that you would like for the camera, such as auto-focus modes, flash modes, and the like. The code snippet shown below demonstrates setting up "continuous picture" auto-focus mode and having the auto-exposure mode engage the flash as needed.

Eventually, you ask the `CaptureRequest.Builder` to `build()` you a `CaptureRequest`, and you pass that to `setRepeatingRequest()` on the `CameraCaptureSession` that is passed into `onConfigure()` of your `CameraCaptureSession.StateCallback`:

```
@Override
public void onConfigured(CameraCaptureSession session) {
  try {
    CaptureRequest.Builder b=
      cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);

    b.addTarget(surface);
    b.set(CaptureRequest.CONTROL_AF_MODE,
              CaptureRequest.CONTROL_AF_MODE_CONTINUOUS_PICTURE);
    b.set(CaptureRequest.CONTROL_AE_MODE,
              CaptureRequest.CONTROL_AE_MODE_ON_AUTO_FLASH);

    // other Builder configuration goes here

    CaptureRequest previewRequest=b.build();

    session.setRepeatingRequest(previewRequest, null, handler);
  }
  catch (CameraAccessException e) {
    // do something
  }
  catch (IllegalStateException e) {
    // do something
  }
}
```

`setRepeatingRequest()` takes three parameters:

- the `CaptureRequest` created by the `Builder`
- an optional `CameraCaptureSession.CaptureCallback` object to be notified about frame captures
- the `Handler` associated with your `HandlerThread`

Note that you will want to hold onto the `CaptureRequest.Builder` that you created here, as you will want it again when it comes time to take a picture.

When you go to `close()` the `CameraDevice`, before you do so, you must also close up the previews. You do this by calling `close()` on the `CameraCaptureSession` and `close()` on your `ImageReader`.

Taking a Picture

At some point, you will want to take a picture. Typically, this is based on user input, though it would not have to be. Taking a picture not only involves telling the camera to capture a picture (typically at a different resolution than the previews), but also to arrange to get that written out to disk somewhere as a JPEG file.

android.hardware.Camera

Taking a photo with a `Camera` is a matter of calling `takePicture()` on the `Camera` object. There are two flavors of `takePicture()`, for which three parameters are in common:

- a `Camera.ShutterCallback`, which will be called the moment the picture is taken, so that you can customize the "shutter" sound
- two `Camera.PictureCallback` objects, for raw (uncompressed) and JPEG photo data, where relatively few devices support raw images using the original camera API

The four-parameter version of `takePicture()` also takes a third `Camera.PictureCallback`, to be called when "a scaled, fully processed postview image is available". This explanation probably means something to somebody, but the author of this book has no idea what it means.

You cannot call `takePicture()` until *after* `startPreview()` has been called to set up a preview pane. `takePicture()` will automatically stop the preview. At some point, if you want to be able to take another photo, you will need to call `startPreview()`

again. Note, though, that you *cannot* call `startPreview()` until after the final compressed photo has been delivered to your `Camera.PictureCallback` object.

Before you call `takePicture()`, you are going to want to adjust the `Camera.Parameters` to configure how the photo should be taken. The primary setting to adjust is the size of the picture to take. Just as you ask `Camera.Parameters` for available preview sizes and choose one, you can call `getSupportedPictureSizes()`, which returns a `List` of `Camera.Size` objects. You can then choose a size and pass its width and height to `setPictureSize()` on the `Camera.Parameters`. Other things to potentially adjust include:

- flash mode (`getSupportedFlashModes()` and `setFlashMode()`)
- focus mode (`getSupportedFocusModes()` and `setFocusMode()`)
- white balance (`getSupportedWhiteBalance()` and `setWhiteBalance()`)
- geo-tagging (`setGpsLatitude()`, `setGpsLongitude()`, `setGpsAltitude()`, etc.)
- JPEG image quality (`setJpegQuality()`)
- and so on

Note that calling `setParameters()` multiple times seems to lead to camera instability. Ideally, you collect all your desired settings from the user up front, then call `setParameters()` once when you set up your preview size. If you need to change parameters, you may wish to consider closing and re-opening the camera.

The `Camera.PictureCallback` will be called with `onPictureTaken()` and will be handed a byte array representing the picture. Typically, you will supply a `PictureCallback` for JPEG images, and so the byte array will represent the photo encoded in JPEG. At this point, you can hand that byte array off to a background thread to write it to disk, upload it to some server, or whatever else you planned to do with the picture.

Note that one thing you cannot readily do with the picture is hand it to another activity. There is a 1MB limit on the size of an `Intent` used with `startActivity()`, and usually the JPEG will be bigger than that. Hence, you cannot readily pass the picture via an `Intent` extra to another activity. If at all possible, use fragments or something else to keep all your relevant bits of UI together in a single activity, rather than try to get the images from activity to activity.

android.hardware.camera2

First, you should attach an `ImageReader.OnImageAvailableListener` instance to your `ImageReader`, using `setOnImageAvailableListener()`. `ImageReader.OnImageAvailableListener` is an interface; you will be called with `onImageAvailable()` when a new image is delivered to the `ImageReader`. We will come back to that `onImageAvailable()` method after quite a bit of additional coding.

Next, given the `CaptureRequest.Builder` you created when you set up the previews, you need to adjust the builder to lock the auto-focus (assuming that auto-focus is enabled):

```
b.set(CaptureRequest.CONTROL_AF_TRIGGER,
        CameraMetadata.CONTROL_AF_TRIGGER_START);
```

At that point, you can `build()` a fresh `CaptureRequest` and call `setRepeatingRequest()` on the `CameraCaptureSession`, to change the previews to switch to a locked focus:

```
captureSession.setRepeatingRequest(b.build(),
              new RequestCaptureTransaction(),
              handler);
```

Here, `RequestCaptureTransaction` is a subclass of `CameraCaptureSession.CaptureCallback`, so you can be notified of how the auto-focus locking is proceeding. You wind up having to implement a fairly convoluted state machine to eventually find out it is time to take a picture... or possibly to ask for a "precapture trigger" to start on the auto-exposure system:

```
private class RequestCaptureTransaction extends CameraCaptureSession.CaptureCallback {
  private final Session s;
  boolean isWaitingForFocus=true;
  boolean isWaitingForPrecapture=false;
  boolean haveWeStartedCapture=false;

  RequestCaptureTransaction(CameraSession session) {
    this.s=(Session)session;
  }

  @Override
  public void onCaptureProgressed(CameraCaptureSession session,
                                  CaptureRequest request, CaptureResult
partialResult) {
```

```
    capture(partialResult);
  }

  @Override
  public void onCaptureFailed(CameraCaptureSession session, CaptureRequest request,
CaptureFailure failure) {
    // TODO: raise event
  }

  @Override
  public void onCaptureCompleted(CameraCaptureSession session, CaptureRequest
request, TotalCaptureResult result) {
    capture(result);
  }

  private void capture(CaptureResult result) {
    if (isWaitingForFocus) {
      isWaitingForFocus=false;

      int autoFocusState=result.get(CaptureResult.CONTROL_AF_STATE);

      if (CaptureResult.CONTROL_AF_STATE_FOCUSED_LOCKED == autoFocusState ||
          CaptureResult.CONTROL_AF_STATE_NOT_FOCUSED_LOCKED == autoFocusState) {
        Integer state=result.get(CaptureResult.CONTROL_AE_STATE);

        if (state == null ||
            state == CaptureResult.CONTROL_AE_STATE_CONVERGED) {
          isWaitingForPrecapture=false;
          haveWeStartedCapture=true;
          capture(s);
        }
        else {
          isWaitingForPrecapture=true;
          precapture();
        }
      }
    }
    else if (isWaitingForPrecapture) {
      Integer state=result.get(CaptureResult.CONTROL_AE_STATE);

      if (state == null ||
          state == CaptureResult.CONTROL_AE_STATE_PRECAPTURE ||
          state == CaptureRequest.CONTROL_AE_STATE_FLASH_REQUIRED) {
        isWaitingForPrecapture=false;
      }
    }
    else if (!haveWeStartedCapture) {
      Integer state=result.get(CaptureResult.CONTROL_AE_STATE);
```

```
    if (state == null ||
        state != CaptureResult.CONTROL_AE_STATE_PRECAPTURE) {
      haveWeStartedCapture=true;
      capture();
    }
  }
}

private void precapture() {
  try {
    b.set(CaptureRequest.CONTROL_AE_PRECAPTURE_TRIGGER,
        CaptureRequest.CONTROL_AE_PRECAPTURE_TRIGGER_START);
    s.captureSession.capture(b.build(), this, handler);
  }
  catch (Exception e) {
    // do something
  }
}

private void capture() {
  try {
    CaptureRequest.Builder captureBuilder=
        cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_STILL_CAPTURE);

    captureBuilder.addTarget(reader.getSurface());
    captureBuilder.set(CaptureRequest.CONTROL_AF_MODE,
        CaptureRequest.CONTROL_AF_MODE_CONTINUOUS_PICTURE);
    captureBuilder.set(CaptureRequest.CONTROL_AE_MODE,
        CaptureRequest.CONTROL_AE_MODE_ON_AUTO_FLASH);

    captureSession.stopRepeating();
    captureSession.capture(captureBuilder.build(),
        new CapturePictureTransaction(), null);
  }
  catch (Exception e) {
    // do something
  }
}
}
```

The author of this book wishes he understood what all this stuff is for.

But, eventually, it will be time to take the picture, represented by the `capture()` method in the above code dump. Here, we create a new `CaptureRequest.Builder`, this time using `TEMPLATE_STILL_CAPTURE` to indicate that we are trying to take a picture. We set up our target (via `addTarget()`) to be the `Surface` from the

`ImageReader`. We re-establish our desired auto-focus and auto-exposure modes. Then, we stop the previews, by calling `stopRepeating()` on the `CameraCaptureSession`, undoing the prior `setRepeatingRequest()` call where we asked for previews. Then, we call `capture()` on the `CameraCaptureSession`, requesting a single-frame capture rather than a repeating request. This, like `setRepeatingRequest()`, takes our `CaptureRequest` from the `Builder`, a `CameraCaptureSession.CaptureCallback` to find out the results of the capture work, and our `Handler`.

The primary job of this `CameraCaptureSession.CaptureCallback` is to restart the previews, in `onCaptureCompleted()`. First, we use the preview edition of the `CaptureRequest.Builder` to undo some of the changes made during the camera capture process. Then, given the original preview `CaptureRequest`, we call `setRepeatingRequest()` again, to get the previews showing once more:

```
@Override
public void onCaptureCompleted(CameraCaptureSession session, CaptureRequest request,
TotalCaptureResult result) {
  try {
    b.set(CaptureRequest.CONTROL_AF_TRIGGER,
        CameraMetadata.CONTROL_AF_TRIGGER_CANCEL);
    b.set(CaptureRequest.CONTROL_AE_MODE,
        CaptureRequest.CONTROL_AE_MODE_ON_AUTO_FLASH);
    s.captureSession.capture(b.build(), null, handler);
    s.captureSession.setRepeatingRequest(previewRequest, null, handler);
  }
  catch (CameraAccessException e) {
    // do something
  }
  catch (IllegalStateException e) {
    // do something
  }
}
```

As part of all of this work, your `onImageAvailable()` method on your `ImageReader.OnImageAvailableListener` will be called when the picture is ready. The recipe for getting your JPEG image looks like this:

```
@Override
public void onImageAvailable(ImageReader imageReader) {
  Image image=imageReader.acquireNextImage();
  ByteBuffer buffer=image.getPlanes()[0].getBuffer();
  byte[] bytes=new byte[buffer.remaining()];

  buffer.get(bytes);
```

```
  image.close();

  // do something with the byte[] of JPEG data
}
```

Here, you are subject to the same sorts of limitations as were described in the section on taking pictures with the original camera API. Notably, that `byte` array may be large, too large to put into an `Intent` extra and pass to another activity.

Recording a Video

Traditional Android video recording is handled via `MediaRecorder`. This means that we need to hand control over the camera from the regular camera API that we are using to `MediaRecorder`, record the video, and then return control back to the camera API (e.g., for previews).

`MediaRecorder` itself then has its own API for configuring the recorder, starting the recording, and stopping the recording.

android.hardware.Camera

To retain the camera access for your app, but allow `MediaRecorder` to take over the camera, call `stopPreview()`, then `unlock()`, on the `Camera` object:

```
camera.stopPreview();
camera.unlock();
```

When the recording is complete, you reverse the process, by calling `reconnect()` and `startPreview()`:

```
camera.reconnect();
camera.startPreview();
```

In between the `unlock()` and `reconnect()` calls is when you use the `MediaRecorder` API.

android.hardware.camera2

This particular combination (video recording with the Android 5.0+ camera API) will be covered in a future edition of this chapter.

Using MediaRecorder

Creating a `MediaRecorder` instance is simple enough: just use the zero-argument constructor.

You then need to tell it what camera to use. With the original camera API, that is a matter of calling `setCamera()` on the `MediaRecorder`, passing in your `Camera` object.

```
MediaRecorder recorder=new MediaRecorder();

recorder.setCamera(camera);
```

Next, call `setAudioSource()` and `setVideoSource()` to indicate where the audio and video to be recorded are coming from. The typical value to use for the audio source is `CAMCORDER`. For the original camera API, you will need to use `CAMERA` as the video source:

```
recorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
recorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
```

Next, you need to configure how the video should be recorded, in terms of things like resolution. The typical approach using the original camera API is to use `setProfile()`, passing in a `CamcorderProfile` to the `MediaRecorder`. You can find out what profiles are supported by calling methods like `hasProfile()` on `CamcorderProfile`. There are some fairly generic profiles, like `QUALITY_HIGH` and `QUALITY_LOW`, and some fairly specific profiles, like `QUALITY_2160P` for 2K video. Not all devices will support all profiles, based on Android version and camera driver capabilities. So, you will need to be responsive to varying cameras and gracefully degrade from the profile you want to a profile that you can get. For example, the following code snippet tries `QUALITY_HIGH`, falls back to `QUALITY_LOW` if `QUALITY_HIGH` is not available, and bails out if neither of those profiles exist:

```
boolean canGoHigh=CamcorderProfile.hasProfile(cameraId,
    CamcorderProfile.QUALITY_HIGH);
boolean canGoLow=CamcorderProfile.hasProfile(cameraId,
    CamcorderProfile.QUALITY_LOW);

if (canGoHigh) {
  recorder.setProfile(CamcorderProfile.get(cameraId,
      CamcorderProfile.QUALITY_HIGH));
}
else if (canGoLow) {
  recorder.setProfile(CamcorderProfile.get(cameraId,
```

```
      CamcorderProfile.QUALITY_LOW));
}
else {
  throw new IllegalStateException(
      "cannot find valid CamcorderProfile");
}
```

Here, `cameraId` is the `int` identifying your open camera.

Then, you can configure:

- the file path to which the resulting video should be written
- the maximum file size you want, after which recording will automatically stop (optional)
- the maximum duration that you want, after which recording will automatically stop (optional)
- a hint for what orientation the video should be recorded in (optional)

```
recorder.
  setOutputFile(new File(getExternalFilesDir(null), FILENAME).getAbsolutePath());
recorder.setMaxFileSize(5000000); // ~5MB max
recorder.setMaxDuration(10000); // ~10 seconds max
recorder.setOrientationHint(90); // rotate output 90 degrees
```

Optionally, you can call `setInfoListener()` and `setErrorListener()`, supplying objects that will be invoked when certain events occur. Notably, if you use `setMaxFileSize()` or `setMaxDuration()`, the `OnInfoListener` object will be notified when recording automatically stops due to reaching one of those limits.

You then call `prepare()`, followed by `start()`, and your video recording will commence:

```
recorder.prepare();
recorder.start();
```

When it comes time to stop the recording manually (e.g., user taps a “stop” button), just call `stop()`, then `release()`, on the `MediaRecorder`.

Configuring the Still Camera

In general, when using the camera classes in Android, you get reasonable defaults for things like focus mode and flash mode. However, what might be reasonable

defaults may not be what the user wants in any given circumstance. Other bits of configuration, like zoom, cannot really be defaulted (other than to "no zoom").

For these, you will need to provide some sort of UI to allow the user to request settings, then apply them as part of your camera implementation. Here, we will focus on applying the configuration.

(and, yes, that was a pun)

Focus Mode

Frequently, a user will want simple autofocus behavior, where the camera attempts to focus on the content centered within the preview. However, in some situations, the user may want autofocus to be disabled, turning the camera into a fixed-focus camera. And there are some specialty focus modes that may be available to you as well, depending upon device and camera API.

Here is how you can set up the camera to use one of those focus modes, for each of the camera APIs.

android.hardware.Camera

The `Camera.Parameters` object has a `getSupportedFocusModes()` method. This returns a `List` of `String` objects, where each value corresponds to a focus mode that is available on this camera (front-facing, rear-facing) on this device. The possible strings are defined as constants on `Camera.Parameters`:

- `FOCUS_MODE_AUTO`
- `FOCUS_MODE_CONTINUOUS_PICTURE`
- `FOCUS_MODE_CONTINUOUS_VIDEO`
- `FOCUS_MODE_EDOF` ("extended depth of field")
- `FOCUS_MODE_FIXED`
- `FOCUS_MODE_INFINITY`
- `FOCUS_MODE_MACRO`

In truth, few devices support all of these. However, every device will support at least one; `getSupportedFocusModes()` is guaranteed to not return `null` and not return an empty `List`.

To choose a focus mode, call `setFocusMode()` on the `Camera.Parameters`, supplying the string of the desired mode. And, of course, you will eventually need to call `setParameters()` on the `Camera`, supplying your modified `Camera.Parameters`.

android.hardware.camera2

Similarly, you can get a list of supported auto-focus modes by calling `get(CameraCharacteristics.CONTROL_AF_AVAILABLE_MODES)` on a `CameraCharacteristics` object tied to your chosen camera. This returns an array of `int` values, instead of a `List` of strings. The possible values are defined as constants on `CameraMetadata`:

- `CONTROL_AF_MODE_AUTO`
- `CONTROL_AF_MODE_CONTINUOUS_PICTURE`
- `CONTROL_AF_MODE_CONTINUOUS_VIDEO`
- `CONTROL_AF_MODE_EDOF` ("extended depth of field")
- `CONTROL_AF_MODE_MACRO`
- `CONTROL_AF_MODE_OFF`

After the user chooses a value, you will need to call `set(CaptureRequest.CONTROL_AF_MODE, ...)` on your `CaptureRequest.Builder`, where `...` is the `int` of the desired focus mode. Note that you will need to do this both for the `CaptureRequest.Builder` for preview frames and for the `CaptureRequest.Builder` used when you take an actual picture. If the user is changing this value while you are already showing the preview, you will need to update the preview behavior, by calling `build()` on the `Builder` to create the `CameraRequest`, then calling `setRepeatingRequest()` to override your previous `CameraRequest` with the new one with the new focus mode. As a result, you tend to want to hang onto your `CameraRequest.Builder` for previews, so you can make these sorts of incremental changes in behavior, without having to create a fresh `Builder` from scratch with all of the desired settings.

Flash Mode

Typically, users want flash when they need flash, due to insufficient ambient lighting. However, once again, they may want specific flash modes instead (definitely flash, definitely *not* flash, etc.).

As with focus modes, you can ask the camera APIs what flash modes are available for a given camera. In this case, though, there is no guarantee of any flash mode

configurability, since not all cameras have flash (and Android considers "off" and "flash does not exist" to be different things). And, once the user has chosen a flash mode, you can configure the camera APIs to use that particular mode.

Of course, the details vary by camera API.

android.hardware.Camera

`Camera.Parameters` has `getSupportedFlashModes()`, which returns a `List` of strings representing the supported flash modes, or `null` if flash modes cannot be configured for this camera. The string values map to constants defined on `Camera.Parameters`:

- `FLASH_MODE_AUTO`
- `FLASH_MODE_OFF`
- `FLASH_MODE_ON`
- `FLASH_MODE_RED_EYE` ("red-eye reduction mode")

There is an additional flash mode, `FLASH_MODE_TORCH`, that will keep the flash during the preview as well as flashing it during the actual act of taking the picture. In truth, this setting is more often used for flashlight apps.

Once the user has chosen a flash mode, you can call `setFlashMode()` on the `Camera.Parameters`, then eventually call `setParameters()` on the `Camera`.

android.hardware.camera2

To find out what flash modes are available for a `camera2` camera, you can call `get(CameraCharacteristics.CONTROL_AE_AVAILABLE_MODES)` on the `CameraCharacteristics` for the camera in question. This returns an array of `int` values, mapping to constants defined on `CameraCharacteristics`:

- `CONTROL_AE_MODE_ON` (which really means "off")
- `CONTROL_AE_MODE_ON_ALWAYS_FLASH`
- `CONTROL_AE_MODE_ON_AUTO_FLASH`
- `CONTROL_AE_MODE_ON_AUTO_FLASH_REDEYE`

Here, `AE` is short for "auto-exposure". `CONTROL_AE_MODE_ON` says that auto-exposure is enabled, just without any flash. There is a separate `CONTROL_AE_MODE_OFF` which totally disables the auto-exposure capability. However, that will screw up auto-focus

and auto-white balance, and so rarely will camera apps want to use `CONTROL_AE_MODE_OFF`.

Once the user chooses the desired flash mode, you can call `set(CaptureRequest.CONTROL_AE_MODE, ...)` on your `CaptureRequest.Builder` object, where `...` is the desired flash mode `int`. You will need to do this both for the preview `Builder` and the `Builder` used when actually taking the picture. If the user is changing this value on the fly, you will need to update the preview behavior, by calling `build()` on the `Builder` to create the `CameraRequest`, then calling `setRepeatingRequest()` to override your previous `CameraRequest` with the new flash-enabled one.

Zoom

Flash and focus modes might be the sort of thing that the user could choose before you start up your camera preview, let alone take a picture. Zoom, on the other hand, is the sort of thing that the user will want to adjust on the fly, based on what they see in the preview.

Hence, your first challenge with implementing a zoom feature is deciding how you want users to indicate that they want to zoom in or out, given that probably most of your screen space is taken up by the preview itself. Options include:

- Float a `SeekBar` over the preview along an edge, where the user can slide the thumb or tap on the bar to move the thumb to indicate an increase or decrease in the zoom
- Use a pinch-zoom gesture, via `ScaleGestureDetector`
- Use some other gesture, such as a vertical swipe, using a `GestureDetector`
- Have a pair of buttons to increase or decrease the zoom

Both Android camera APIs have the notion of a numeric zoom level. The bottom end of the zoom range is either 0 (for the classic camera API) or 1 (for the `camera2` API). The top end is found by from the camera APIs. Your job will be to convert whatever input signals you get from the user into a zoom level, then update the camera settings to zoom to that setting.

The code segments shown in this section assume that your input is giving you a zoom level in the 0-100 range, such as via a `SeekBar` with the default maximum value.

android.hardware.Camera

`Camera.Parameters` offers several methods related to zoom.

The big one is `isZoomSupported()`. `false` means that the camera does not offer any sort of zoom (digital or optical). You might use that to disable your zoom input option, so as not to offer something to the user that will not work. Few devices will return `false`, though.

Assuming `isZoomSupported()` is `true`, then `getMaxZoom()` will tell you the highest possible zoom value. Your overall range of zoom values will be from 0 to this maximum.

If you are using some form of user input that only indicates incremental changes in zoom (e.g., buttons for zoom in and zoom out), you can use `getZoom()` to find out the current zoom value. You can then increment or decrement that value and check your new value against the ends of the range (0 and `getMaxZoom()`) to ensure that it is valid.

Given a new zoom value, you have two choices for applying it:

- `setZoom()` on `CameraParameters` does a "smash cut", jumping to the new zoom value immediately upon applying those parameters to the camera via `setParameters()`.
- `startSmoothZoom()` on `Camera` will "animate" the zoom change from the current to the new value over a period of a second or two. However, not all devices support this. Call `isSmoothZoomSupported()` on the `Camera.Parameters` to see if smooth zoom is available to you.

The following code snippet takes a zoom level from 0 to 100 and zooms the camera, assuming zoom is supported:

```
@Override
public boolean zoomTo(Camera camera, int zoomLevel) {
  Camera camera=descriptor.getCamera();
  Camera.Parameters params=camera.getParameters();
  int zoom=zoomLevel*params.getMaxZoom()/100;
  boolean result=false;

  if (params.isSmoothZoomSupported()) {
    camera.setZoomChangeListener(this);
    camera.startSmoothZoom(zoom);
    result=true;
```

```
  }
  else if (params.isZoomSupported()) {
    params.setZoom(zoom);
    camera.setParameters(params);
  }

  return(result);
}
```

You will notice that if `isSmoothZoomSupported()` returns `true`, we not only call `startSmoothZoom()`, but we also call `setZoomChangeListener()`. This registers a listener to find out about how the smooth zoom is progressing. In particular, you should disable further changes to the zoom until the smooth zoom process completes. Your `OnZoomChangeListener` will be called with `onZoomChange()` for each incremental change in the zoom from start to finish, with `stopped` set to `true` when we are done with the smooth zoom operation:

```
@Override
public void onZoomChange(int zoomValue, boolean stopped,
                         Camera camera) {
  if (stopped) {
    // do something
  }
}
```

If you need to stop the smooth zoom before completion, there is a `stopSmoothZoom()` method on `Camera` that you can call. For example, instead of disabling zoom controls, you might stop the current smooth zoom operation if the user chooses a new zoom level, then start a fresh smooth zoom operation to the newly-requested level.

android.hardware.camera2

(the author would like to thank Daniel Albert for helping with this section)

On the surface, the `camera2` API works much the same: you find out the maximum zoom value, translate your user input into the valid zoom value range (this time, from `1.0f` to the maximum), and then update the camera for that zoom value.

However, that last step is substantially different than before.

For digital zoom, rather than saying "zoom in to this value", we say "crop the camera inputs to this rectangle, and expand that rectangle to fill the preview or the picture". This is rather more complex, albeit with potentially more power.

To find out the maximum digital zoom value, call `get(CameraCharacteristics.SCALER_AVAILABLE_MAX_DIGITAL_ZOOM)` on the `CameraCharacteristics` for the camera in question. That will be a float value. `1.0f` would indicate that the camera cannot perform digital zoom. The range of possible digital zoom values is from `1.0f` to whatever the maximum is.

So, the first part of this edition of `zoomTo()` normalizes a 0-100 integer into a `float` representing the zoom value:

```
@Override
public boolean zoomTo(String cameraId,
                      CaptureRequest.Builder previewRequestBuilder,
                      CameraCaptureSession captureSession,
                      int zoomLevel) {
  try {
    final CameraCharacteristics cc=
      mgr.getCameraCharacteristics(cameraId);
    final float maxZoom=
      cc.get(
        CameraCharacteristics.SCALER_AVAILABLE_MAX_DIGITAL_ZOOM);

    // if <=1, zoom not possible, so eat the event
    if (maxZoom>1.0f) {
      float zoomTo=1.0f+((float)zoomLevel*(maxZoom-1.0f)/100.0f);

      zoomRect=cropRegionForZoom(cc, zoomTo);

      previewRequestBuilder
        .set(CaptureRequest.SCALER_CROP_REGION, zoomRect);
      previewRequest=previewRequestBuilder.build();
      captureSession.setRepeatingRequest(previewRequest,
        null, handler);
    }
  }
  catch (CameraAccessException e) {
    // ummm... do something
  }

  return(false);
}
```

Given a zoom value, we need to determine the `Rect` that represents the subset of the field of vision that we want to zoom into. The following algorithm zooms into the center of the field:

```
private static Rect cropRegionForZoom(CameraCharacteristics cc,
                                      float zoomTo) {
  Rect sensor=
    cc.get(CameraCharacteristics.SENSOR_INFO_ACTIVE_ARRAY_SIZE);
  int sensorCenterX=sensor.width()/2;
  int sensorCenterY=sensor.height()/2;
  int deltaX=(int)(0.5f*sensor.width()/zoomTo);
  int deltaY=(int)(0.5f*sensor.height()/zoomTo);

  return(new Rect(
    sensorCenterX-deltaX,
    sensorCenterY-deltaY,
    sensorCenterX+deltaX,
    sensorCenterY+deltaY));
}
```

That `Rect` then gets used:

- Immediately, via a call to `set()` on the `CaptureRequest.Builder`, to set the `SCALER_CROP_REGION`. That `Builder` then is used to re-establish the preview repeating capture request.
- At the point in time when the user requests to take a picture. We will need to call `set()` on *that* `CaptureRequest.Builder`, to reproduce the same zoom.
- Later, if we have to set up the preview capture request again and we still want this zoom value taken into account.

Note that this does not cover optical zoom. On Android 5.0, that is handled as available focal lengths. You can get the list of available focal lengths by requesting `LENS_INFO_AVAILABLE_FOCAL_LENGTHS` from the `CameraCharacteristics`. Setting `LENS_FOCAL_LENGTH` on a `CaptureRequest.Builder` will shift the camera's focal length as requested. This may take a moment, as optical zoom usually requires mechanical changes in the camera configuration. The `LENS_STATE` (on `CaptureResult`) will be reported as `MOVING` while the focal length is changing, or `STATIONARY` once the focal length has reached the requested value.

And Now, The Problems

Of course, taking pictures is not nearly this simple. The preceding sections glossed over all sorts of problems that you will run into in practice when trying to implement these APIs. The following sections outline a few of those problems, particularly ones that will affect both camera APIs.

Choosing a Preview Size

Camera drivers are capable of delivering preview images to your preview surface in one of several resolutions. You have to sift through a roster of resolutions and choose one.

Your gut instinct might be to choose the highest-available resolution. After all, that should result in the highest-quality previews. However, this can be wasteful, if the preview images are significantly bigger than your preview surface. Plus, the larger the preview frames, the slower the camera driver will be to deliver them, reducing your possible frames-per-second (fps) for the previews. You might instead elect to choose the largest preview that is smaller than the surface, or some algorithm like that.

Previews and Aspect Ratios

Compounding the problem of choosing preview sizes is that the resolutions of available preview sizes bear no relationship at all to the size of your preview surface. After all, you might have a `TextureView` that fills the screen, or you might have a `TextureView` that is rather tiny. That is up to you from a UI design standpoint; the camera driver is oblivious to such considerations.

In particular, the aspect ratios (width divided by height) of the preview frames do not necessarily have to match the aspect ratio of your preview surface. For example, few camera drivers support square previews, yet for aesthetic reasons you might be aiming for a square preview surface.

You have two main approaches for dealing with this: letterboxing and cropping.

Letterboxing is where your preview frames retain their aspect ratio, but do not fill up all the available space in the preview surface. Instead, part of the preview surface is unused. For example, if your preview surface is square, and your preview frames have a landscape aspect ratio (width is greater than the height), letterboxing would

show the landscape aspect ratio within the square box of the preview surface, with black bars for the unused portion of the square's height. Typically, using gravity, you try to have the preview frames be centered and the unused portion of the surface be split to either side of the frames.

If you want to fill the preview surface, then letterboxing is not a viable option. However, if you just take the preview frames and try to put them into the surface, the surface will stretch the frames to fit the surface. If the aspect ratio of the frames is significantly different than is the aspect ratio of the surface, the subject matter in the preview will seem significantly stretched, either vertically or horizontally.

The trick to deal with this, on API Level 14+ (with graphics acceleration enabled, as is the default), is to have the surface be *bigger* than what you really want, but then to have something overlapping the surface and causing it to be visually cropped. You have your new, larger surface match the aspect ratio of the preview frames, so there is no stretching. However, now what the user sees in your preview surface may differ substantially from what winds up in the picture or video, as you are cropping off portions that do not fit your preview surface, where those cropped areas might well show up in final output.

Choosing a Picture or Video Size

Choosing a picture or video size is reminiscent of choosing a preview size. While many cases will call for as high of a resolution as you can muster, some use cases will lead you towards choosing a lower resolution. For example, situations requiring a rapid upload of the resulting media might select a lower resolution, as that will reduce the file size and make the upload process that much faster.

Also, bear in mind that the aspect ratio of the available picture or video sizes do not necessarily match the aspect ratio of *either* the preview frames *or* your preview surface. Emphasize to your users that the preview surface is for *aiming* the camera; what actually gets recorded may be somewhat different in scope but should be centered on the same spot.

Picture Orientation

Your app may wish to take pictures in both landscape and portrait modes. However, the camera drivers are designed around taking pictures in landscape, particularly for rear-facing cameras.

You can hint to the camera driver what orientation you think the resulting picture should have, such as via `setRotation()` on the `Camera.Parameters` in the original camera API. However, as the documentation for that method states:

> The camera driver may set orientation in the EXIF header without rotating the picture. Or the driver may rotate the picture and the EXIF thumbnail. If the Jpeg picture is rotated, the orientation in the EXIF header will be missing or 1 (row #0 is top and column #0 is left side).

Many camera drivers take the approach of leaving the image alone and setting the `Orientation` EXIF header. That header tells image viewers to rotate the image. Unfortunately, not all image viewers or image decoding libraries pay attention to this. Notably, *Android* usually does not pay attention to this, as `BitmapFactory` ignores this EXIF header. As a result, when you go to load in your own picture that you took, your result may come out mis-oriented.

You have two major choices:

1. Put more smarts in any logic that you are using to display images that you take with the camera, where you read the EXIF headers yourself and you arrange to rotate the image as needed, perhaps by rotating the `ImageView` you are using to show the image.
2. As part of post-processing the image before saving it, you rotate the image based upon what is in the EXIF header, and save the image with the proper rotation and no EXIF header. This has the advantage of making the image "correct" for all image viewers. However, rotating full-resolution photos is rather memory-intensive and slow. Using NDK code, such as this library, may be able to help.

Storage Considerations

Bear in mind that if you wish to save pictures or videos in common locations on external storage, such as the standard location for digital camera output (`Environment.DIRECTORY_DCIM`), you will need the `WRITE_EXTERNAL_STORAGE` permission on all relevant API levels. As of Android M, this is a `dangerous` permission handled via the runtime permission system, so you will need to have the `<uses-permission>` element in the manifest *and* ask the user for that permission at runtime.

Also, files written out to external storage will not be picked up immediately by `MediaStore`, and so "gallery" and related apps that rely upon the `MediaStore` will not

see your pictures or videos. You can use `MediaScannerConnection` to proactively have the `MediaStore` add your newly-created files to the index, as was covered earlier in the book.

Configuration Changes

Opening and closing a camera each takes a fair amount of time. As a result, if your app wants to support taking pictures and videos in either portrait or landscape, this is a case where you will want to strongly consider using a retained fragment to hold onto your `Camera` (or combination of `CameraManager` and `CameraDevice`) across a configuration change. That way, Android will not destroy and recreate the fragment, and you can keep the camera open during the change.

Camera Peeking Attacks

(NOTE: this section is based upon a blog post from the author)

A research paper points out an interesting Android attack vector, resulting in a possible leak of private information. The paper's authors refer to it as the "camera peeking" attack.

An Android camera driver can only be used by one app at a time. The attack is simple:

- monitor for when an app that might use the camera for something important comes to the foreground
- at that point, start watching for the camera to become unavailable
- once the camera is unavailable, then available again, grab the camera and take a picture, in hopes that the camera is still pointing at the private information

The example cited by the paper's authors is to watch for a banking app taking a photo of a check, to try to take another photo of the check to send to those who might use the information for various types of fraud.

Polling for camera availability is slow, simply because the primary way to see if the camera is available is to open it, and that takes hundreds of milliseconds. The paper's specific technique helped to minimize the polling, by knowing when the right activity was in the foreground and therefore the camera was probably already in use. Then, it would be a matter of polling until the camera is available again and

taking a picture. Even without the paper's specific attack techniques, this general attack is possible, and there may be more efficient ways to see if the camera is in use.

On the other hand, the defense is simple: if your app is taking pictures, and those pictures may be of sensitive documents, ask the user to point the camera somewhere else before you release the camera. So long as you have exclusive control over the camera, nothing else can use it, including any attackers.

A sophisticated implementation of this might use image-recognition techniques to see, based upon preview frames plus the taken picture, if the camera is pointing somewhere else. For example, a banking app offering check-scanning might determine if the dominant color in the camera field significantly changes, as that would suggest that the camera is no longer pointed at a check, since checks are typically fairly monochromatic.

Or, just ask the user to point the camera somewhere else, then close the camera after some random number of seconds.

General-purpose camera apps might offer an "enhanced security" mode that does this sort of thing, but having that on by default might annoy the user trying to take pictures at the zoo, or at a sporting event. However, document-scanning apps might want to have this mode on by default, and check-scanning apps might simply always use this mode.

Media Routes

Android can send audio and video to a variety of places, such as:

- Bluetooth headsets or headphones
- External displays, like a TV or monitor
- External devices that themselves play back media, such as a Chromecast

There is a common API for determining which of these "places" are available and allowing the user to choose which of these "places" should be used for a given bit of media. This common API centers around a `MediaRouter`, which is the focus of this chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of the book. In addition, you should read the chapters on advanced action bar techniques and the `appcompat-v7` action bar backport.

Terminology

First, we need to establish some common ground in terms of..., well, terms.

Media

In this chapter, "media" refers to audio or video. This includes both media that may be stored on the device as well as media that may be streamed from some other source, frequently over the Internet.

Route

A route indicates where media should be played. There are three categories of routes that concern us:

- Where should we be playing live audio, in terms of speakers or headphones or other things connected to the device?
- Where should we be playing live video: on the device's own screen or on some other screen connected via a cable?
- Is there any sort of "remote playback" device available, such as a Chromecast, that can play back media on its own under our direction, rather than requiring our own app to play back the media itself?

MediaRouter

`MediaRouter` is the name of a class (actually, two classes) that know what routes are possible given the current environment and what routes are selected for the different categories (by default or by user choice).

A Tale of Two MediaRouters

`MediaRouter` and its related classes represent a curious API. There are two versions of the `MediaRouter` class related support classes that will concern you as a developer.

android.media

`MediaRouter` debuted in Android in API Level 16, through classes added to the `android.media` package. This version of `MediaRouter` can work with live audio and live video routes, but not the Chromecast-style remote playback routes.

`android.media` also contains other classes that pertain to routes, such as `MediaRouteActionProvider`, a way to allow the user to choose media routes via an action bar item. The version of these classes in `android.media` work with native API Level 11 versions of the action bar and fragments.

android.support.v7.media

In 2013, an update to the Android Support package was released that contained another version of `MediaRouter` and kin, in `android.support.v7` packages. These are contained in a dedicated Android library project that you can add to your app, found

in the extras/android/support/v7/mediarouter directory of your Android SDK installation, if you have a current Android Support package installed.

While the native version of MediaRouter is a system service — obtained via getSystemService() – the v7 version of MediaRouter is a singleton, obtained from a static getInstance() method on the MediaRouter class.

The good news is that this updated version of MediaRouter can work with all three categories of routes, including the Chromecast-style remote playback routes.

However, the bad news is that the v7 version of MediaRouter's support classes only support the Android Support backports of fragments and the action bar. This requires you to inherit from ActionBarActivity and use the v4 version of Fragment and kin. This is a rather annoying limitation, considering that many developers have specifically started dropping support for older API levels to be able to *avoid* using this backport.

Attaching to MediaRouter

To be able to take advantage of all that MediaRouter has to offer, we need to obtain an instance of it and connect to that instance, via method calls and registering callbacks.

Getting a MediaRouter Instance

To get an instance of the android.support.v7.media.MediaRouter flavor of MediaRouter, call getInstance() on MediaRouter.

This is in contrast to the android.media.MediaRouter variant, which is a system service, obtained by calling getSystemService().

Note that the android.support.v7.media.MediaRouter flavor is global for your process, but weakly held from a garbage collection standpoint. You need to ensure that you hold onto your instance of MediaRouter as long as you need it. Once your application code lets go of the MediaRouter instance, it becomes eligible for garbage collection, disposing of any registered callbacks and such along the way.

Working with Routes

`MediaRouter` has a `getSelectedRoute()` method that returns the media route chosen by the user, or the overall default if the user has not yet had a chance in your app to choose a route. This method returns a `MediaRouter.RouteInfo` object, containing details about the route. In particular, you can call `supportsControlCategory()` to determine if the route is a live audio route, a live video route, or a remote playback route, so you can take advantage of it accordingly.

There is also `getDefaultRoute()`, which, as the name suggests, returns the `MediaRouter.RouteInfo` instance that is the overall default for your app.

You can call `getRoutes()` to obtain a list of all routes known at the present time. You might use this to allow the user to choose a route, though `MediaRouteActionProvider` is generally a better choice, as will be seen later in this chapter.

Given that you have a `MediaRouter.RouteInfo` instance from somewhere, you can call `selectRoute()` to make this route the active one, replacing whatever the previously-selected route was.

Registering a Callback

You can also call `addCallback()` to provide a `MediaRouter.Callback` instance that will be invoked at various points in time based on the changes in media routes. `addCallback()` also takes a `MediaRouteSelector`, which describes what sorts of routes you are interested in. We will examine `MediaRouteSelector` in greater detail in the coverage of `MediaRouteActionProvider` later in this chapter.

There are two flavors of `addCallback()`. Both take the `MediaRouteSelector` and the `MediaRouter.Callback`, but one also takes an `int` supplying flags to control the behavior of `addCallback()`. One flag of particular importance is `CALLBACK_FLAG_REQUEST_DISCOVERY`. This tells `MediaRouter` to not only set up the callback, but to attempt to find new routes previously unknown to it. Mostly, this is for remote playback routes, which require network I/O to find and are not necessarily known if not specifically scanned for.

`MediaRouter.Callback` is a class, not an interface. You create your own subclass of `MediaRouter.Callback` and override the callback methods that interest you. Some noteworthy callback methods include:

- `onRouteAdded()` and `onRouteRemoved()`, which are called when routes are newly detected or have been lost, such as when a user plugs in or unplugs an HDMI cable from the device
- `onRouteSelected()` is called when a new route is selected, either by the user (e.g., via `MediaRouteActionProvider`) or by you (e.g., via `selectRoute()`)
- `onRouteUnselected()` is also called when a new route is selected, but in this case, you are notified about the *old* route being unselected

When you are done with the callback, call `removeCallback()` on the `MediaRouter`, passing in the same `MediaRouter.Callback` instance you supplied to `addCallback()`.

We will see examples of using `MediaRouter.Callback` in the next section.

User Route Selection with MediaRouteActionProvider

To give the user some measure of control over where media is played, you can add a `MediaRouteActionProvider` to your action bar. This will add a button that, when tapped, will allow the user to choose routes of relevance to your app (live audio, live video, remote playback).

However, this does not really work the way you (or the user) might expect, simply because some routes are automatically applied by the OS. Depending upon what the Android device is connected to will determine what routes are automatically applied and which ones the user can choose via `MediaRouteActionProvider`. For example, while Android will route live video to an HDMI-connected external display automatically, the user must opt into connecting to a Chromecast for remote playback capability.

This section outlines how to use `MediaRouteActionProvider` and what the user will see for various circumstances. Most of the sections will be focusing on the MediaRouter/ActionProvider sample project.

The Basic Project and Dependencies

The project has dependency on the `mediarouter` Android library project. Projects that need `mediarouter` will need to have access to the Android Support library from the SDK Manager and follow the instructions to add it to your project. Since `mediarouter-v7` depends upon `appcompat-v7`, you will need both library projects.

The `appcompat-v7` backport of the action bar requires that your activities use a theme extending from `Theme.AppCompat`. Hence, we have a `res/values/styles.xml` resource that defines `AppTheme` in the context of `Theme.AppCompat.Light.DarkActionBar`:

```
<resources>

  <style name="AppBaseTheme" parent="@style/Theme.AppCompat.Light.DarkActionBar"></style>

  <style name="AppTheme" parent="AppBaseTheme">
    <!-- All customizations that are NOT specific to a particular API-level can go here. -->
  </style>

</resources>
```

(from MediaRouter/ActionProvider/app/src/main/res/values/styles.xml)

And our `<activity>` in the manifest, pointing to `MainActivity`, refers to that theme:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.mrap"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="18"/>

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="com.commonsware.android.mrap.MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from MediaRouter/ActionProvider/app/src/main/AndroidManifest.xml)

The Menu Resource

Since `MediaRouteActionProvider` is an action provider, we can add it to our action bar via an `actionProviderClass` attribute in a menu resource. And, since the Google implementation of `MediaRouteActionProvider` works with the `appcompat-v7` action bar backport, we specifically need to use the `appcompat-v7` approach to adding `actionProviderClass`, putting it in our app's custom XML namespace:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/route_provider"
    android:title="@string/route_provider_title"
    app:actionProviderClass="android.support.v7.app.MediaRouteActionProvider"
    app:showAsAction="always"/>

</menu>
```

(from MediaRouter/ActionProvider/app/src/main/res/menu/main.xml)

Initializing the MediaRouter and Selector

Our activity (`MainActivity`) is an `AppCompatActivity` subclass, following the rules for using the `appcompat-v7` action bar backport:

```
package com.commonsware.android.mrap;

import android.os.Bundle;
import android.support.v4.view.MenuItemCompat;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.app.MediaRouteActionProvider;
import android.support.v7.media.MediaControlIntent;
import android.support.v7.media.MediaRouteSelector;
import android.support.v7.media.MediaRouter;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
  private MediaRouteSelector selector=null;
  private MediaRouter router=null;
  private TextView selectedRoute=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    selectedRoute=(TextView)findViewById(R.id.selected_route);
```

```
    router=MediaRouter.getInstance(this);
    selector=
        new MediaRouteSelector.Builder().addControlCategory(MediaControlIntent.CATEGORY_LIVE_AUDIO)
                                        .addControlCategory(MediaControlIntent.CATEGORY_LIVE_VIDEO)
                                        .addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)
                                        .build();

  }

  @Override
  public void onStart() {
    super.onStart();

    router.addCallback(selector, cb,
                       MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);
  }

  @Override
  public void onStop() {
    router.removeCallback(cb);

    super.onStop();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);

    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

    provider.setRouteSelector(selector);

    return(true);
  }

  private MediaRouter.Callback cb=new MediaRouter.Callback() {
    @Override
    public void onRouteSelected(MediaRouter router,
                                MediaRouter.RouteInfo route) {
      selectedRoute.setText(route.toString());
    }
  };
}
```

(from MediaRouter/ActionProvider/app/src/main/java/com/commonsware/android/mrap/MainActivity.java)

In `onCreate()` we obtain an instance of `MediaRouter`. More specifically, we obtain an instance of `android.support.v7.media.MediaRouter`.

We also will need a `MediaRouteSelector` instance. `MediaRouteSelector` expresses rules for what sorts of media routes we are interested in. The simplest way to set up a `MediaRouteSelector` is to use the `MediaRouteSelector.Builder` inner class, which follows the fluent API style of other Android `Builder` classes (e.g., `Notification.Builder`, `AlertDialog.Builder`). Here, we call

addControlCategory() three times, indicating three categories of routes that we are interested in:

- MediaControlIntent.CATEGORY_LIVE_AUDIO
- MediaControlIntent.CATEGORY_LIVE_VIDEO
- MediaControlIntent.CATEGORY_REMOTE_PLAYBACK

Calling build() on the resulting Builder gives us our MediaRouteSelector, which we will use elsewhere in the activity.

Configuring the ActionProvider

In onCreateOptionsMenu() of MainActivity, we inflate our menu resource and pull out the MediaRouteActionProvider. To obtain an action provider from the appcompat-v7 action bar, the simplest solution is to use the MenuItemCompat helper class from the Android Support package, calling its static getActionProvider() method. This will work both with the appcompat-v7 backport of the action bar *and* with the native API Level 11+ action bar, though you do not need to use MenuItemCompat for the latter if you do not want.

We then call the setRouteSelector() method on our MediaRouteActionProvider instance, passing in the MediaRouteSelector we configured back in onCreate(). This tells the action provider what routes the user should be able to configure. In our case, that is all three major categories of routes (live audio, live video, and remote playback).

Registering for Route Changes

Interestingly enough, that is insufficient to make the MediaRouteActionProvider work. We *also* need to register a MediaRouter.Callback with the MediaRouter, to be informed about events related to media routes. Our cb private data member is an instance of an anonymous inner class extending MediaRouter.Callback, overriding the onRouteSelected() method. This method will be called whenever a new route is selected, telling us the MediaRouter.RouteInfo of the newly-selected route. In our case, we just update a TextView that is our activity's UI with the details of that route, courtesy of calling toString() on the RouteInfo object.

To inform MediaRouter about our desire for such callbacks, we need to call addCallback() on the MediaRouter, and later on call removeCallback() when we no longer need to know about such events. In MainActivity, these steps are done in onStart() and onStop(), respectively.

Note that we provide the `CALLBACK_FLAG_REQUEST_DISCOVERY` flag in the `addCallback()` method, to trigger a search for any Chromecast or other remote playback-capable devices that can serve as media routes.

The Results

Running this on an emulator is largely pointless, as emulators do not emulate media routes.

Running this on a device will give varying results, depending upon what other media-related accessories are available to that device. If there are no user-selectable media routes available, the `MediaRouteActionProvider` is marked as invisible, so the user does not see the icon and perhaps get confused by why tapping on it has no effect.

However, our `TextView` will show some initial route that was chosen by the device:

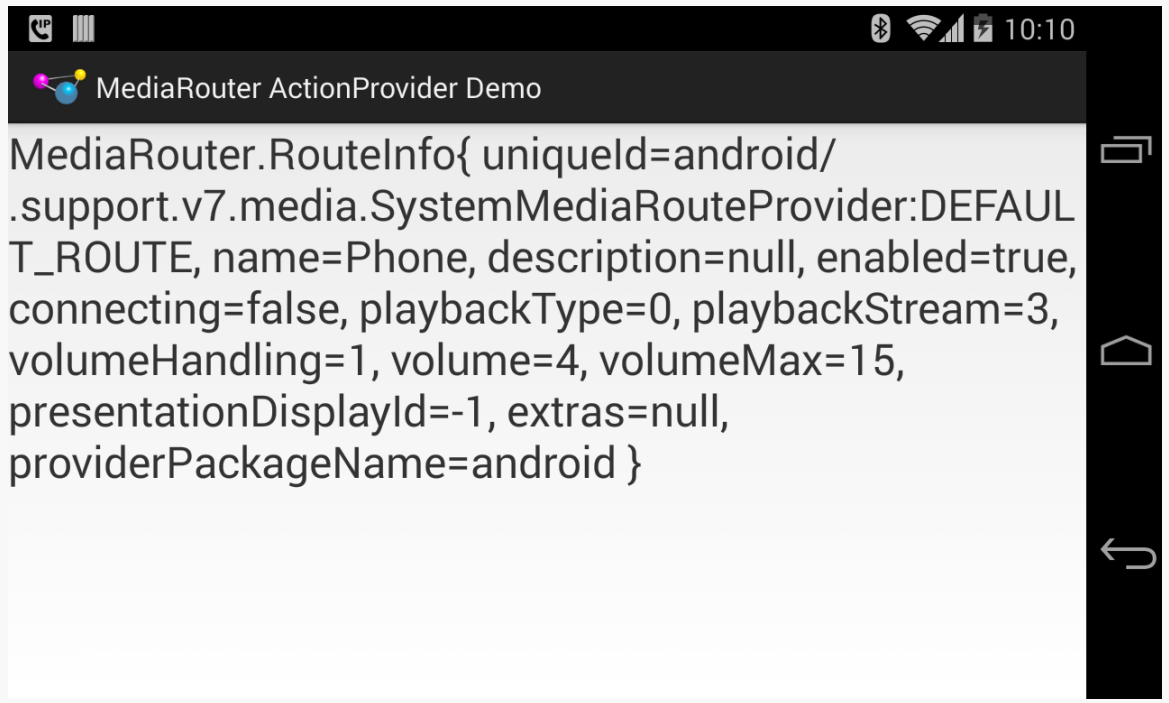


Figure 820: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Default Route

Live Audio Routes

If you launch the demo with some form of external headset or speakers attached, such as via Bluetooth, you will see the route for that is automatically selected:

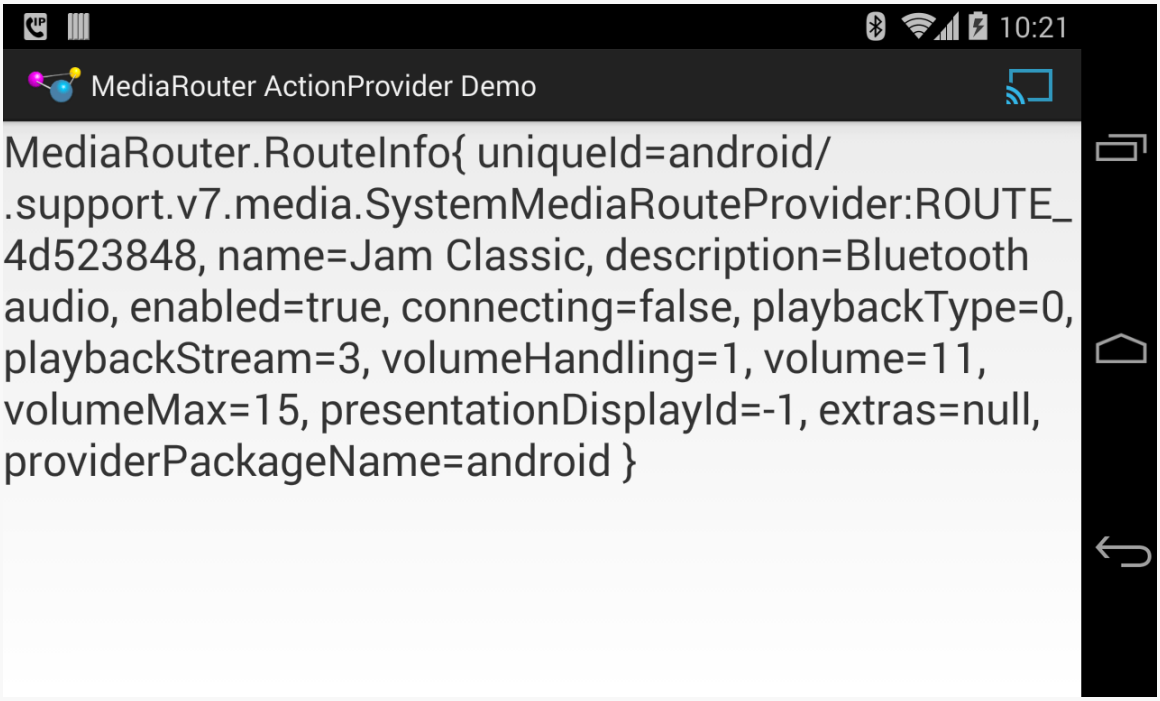


Figure 821: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Live Audio Route

The `MediaRouteActionProvider` appears, with a blue highlight, indicating an active selected route. More importantly, the blue highlight indicates that the route is configurable by tapping on it to bring up a dialog:

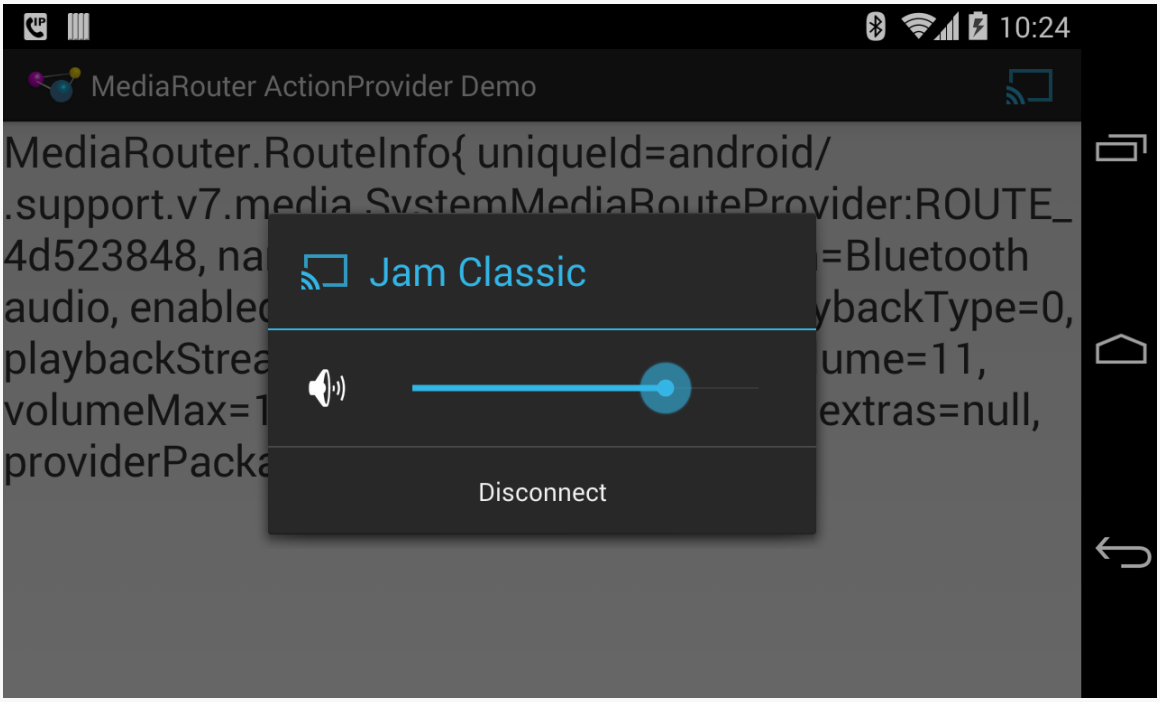


Figure 822: MediaRouter ActionProvider Demo, on a Nexus 4, Live Audio Route Configuration

Here, we can adjust the volume, plus disconnect from the route. Disconnecting shows our `MediaRouteActionProvider` with the default white highlight:

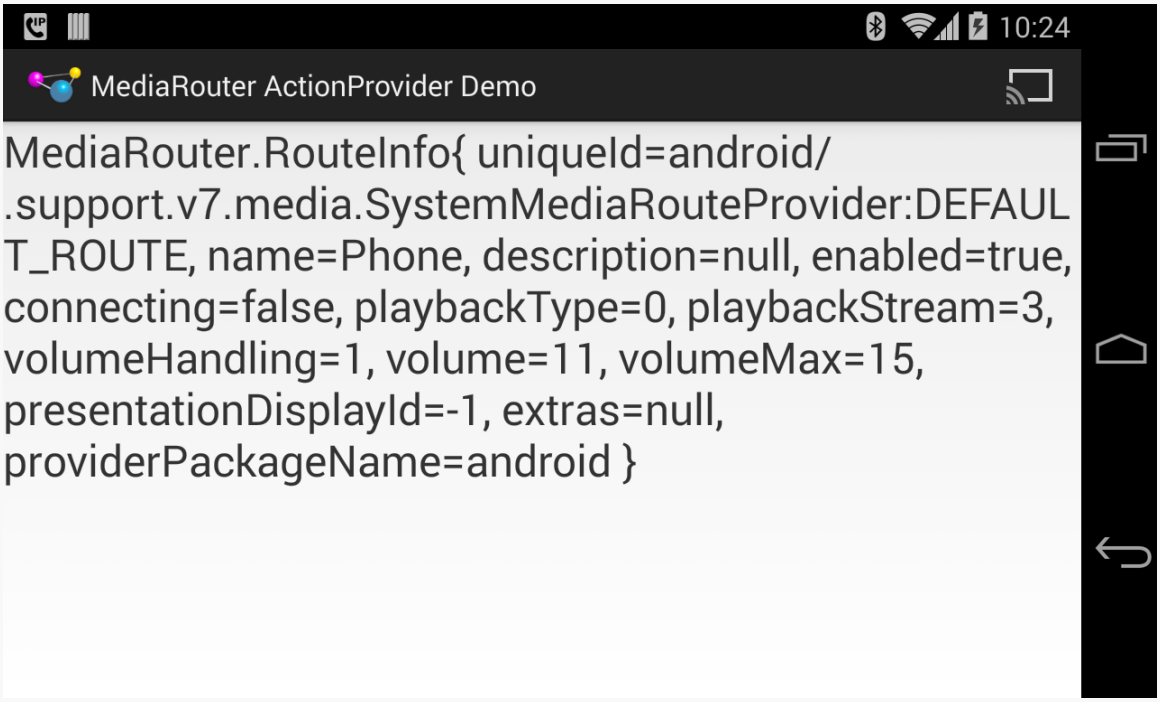


Figure 823: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Default Route and Provider

The white highlight means that there are possible routes, though none in use. Tapping the icon brings up a connection dialog:

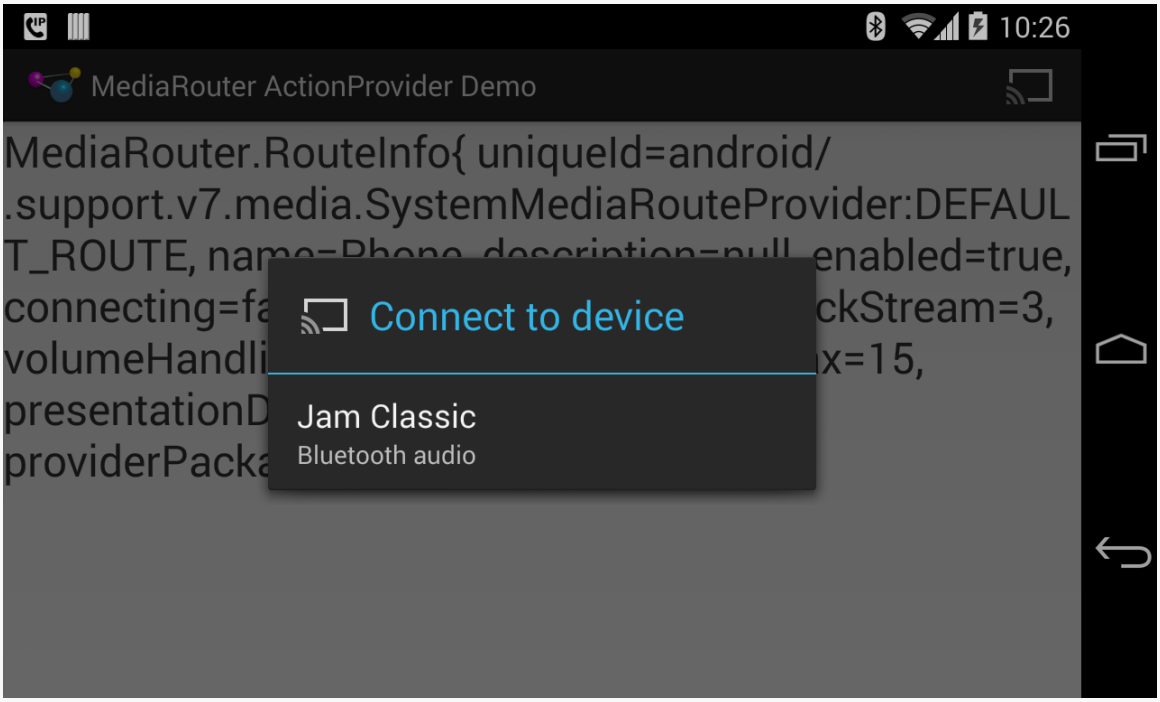


Figure 824: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Available Routes

Live Video Routes

If you launch the demo with some form of external display attached — HDMI, MHL, SlimPort, etc. — you still will not see the `MediaRouteActionProvider`, as live video routes are automatically selected, at least if there is only one such route.

However, `onRouteSelected()` will still be called as part of starting up the activity, so the `TextView` will reflect the live video route:

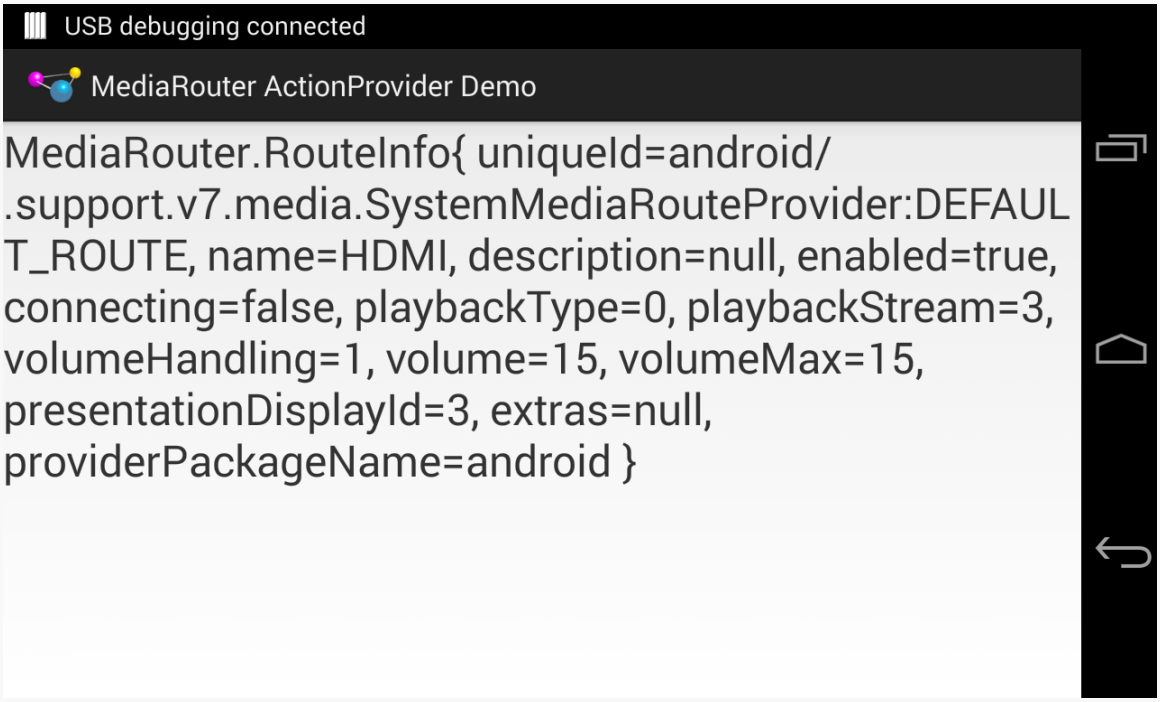


Figure 825: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Live Video Route

Remote Playback Routes

Since the user has to opt into remote playback media routes, the `MediaRouteActionProvider` *will* appear if you configure it to show such routes and a route is available:

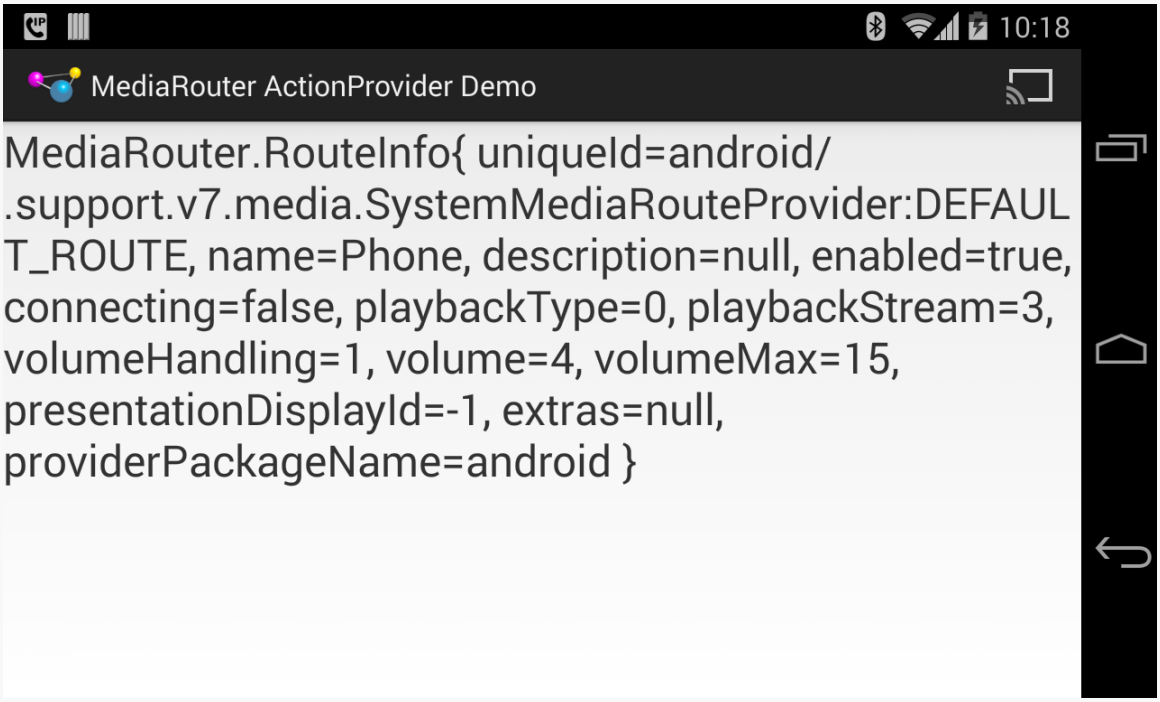


Figure 826: MediaRouter ActionProvider Demo, on a Nexus 4, Showing ActionProvider

The `MediaRouteActionProvider`, when tapped, will pop up a dialog of available routes that the user can select:

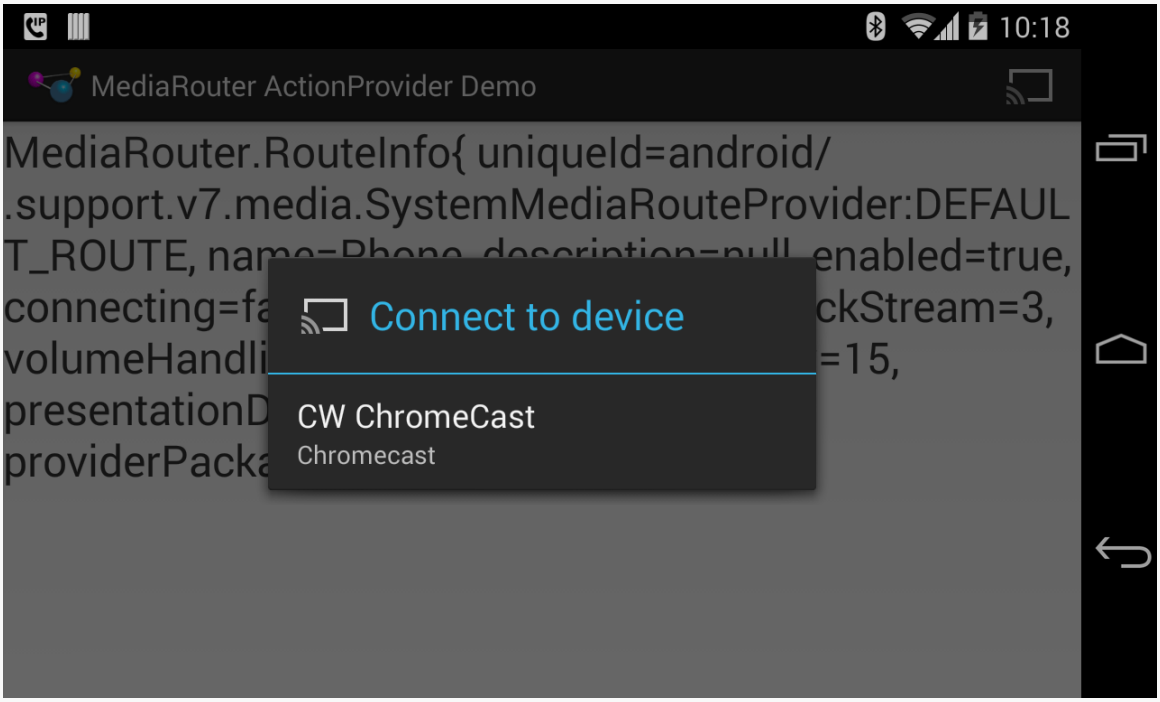


Figure 827: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Available Chromecast Route

Note that if the device has both a Bluetooth audio connection *and* access to a remote playback route (like a Chromecast), and you requested both live audio and remote playback routes, then the route selection dialog could have multiple choices:

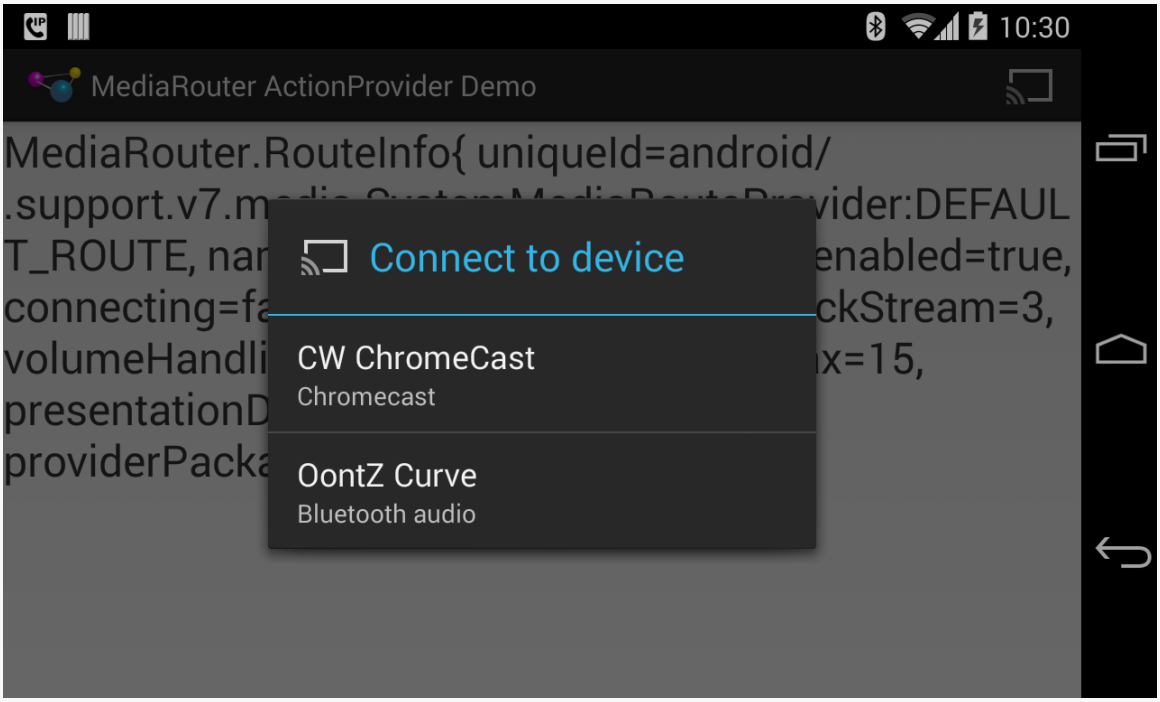


Figure 828: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Multiple Available Routes

If the user chooses a route from the dialog, our `onRouteSelected()` method will be called to reflect the new selection:

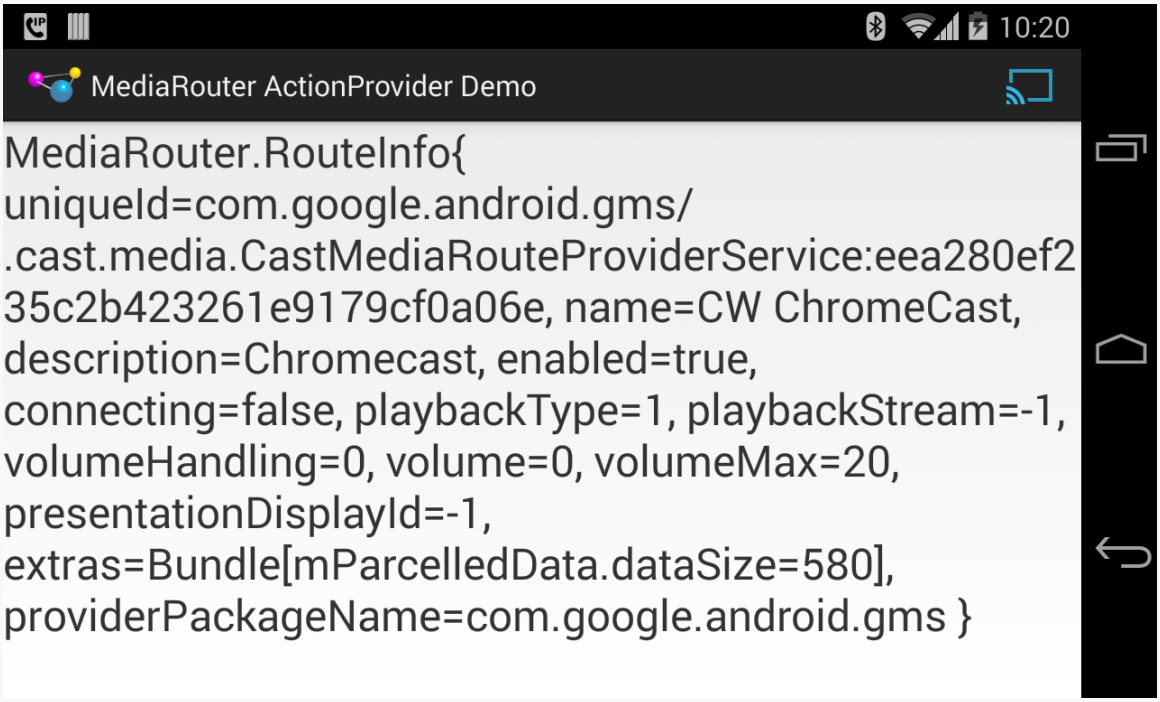


Figure 829: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Selected Chromecast Route

Also note that the `MediaRouteActionProvider` color changes from white to blue, indicating an altered route.

Tapping the action provider again pops up a dialog to control the volume of the route, plus a “Disconnect” button:

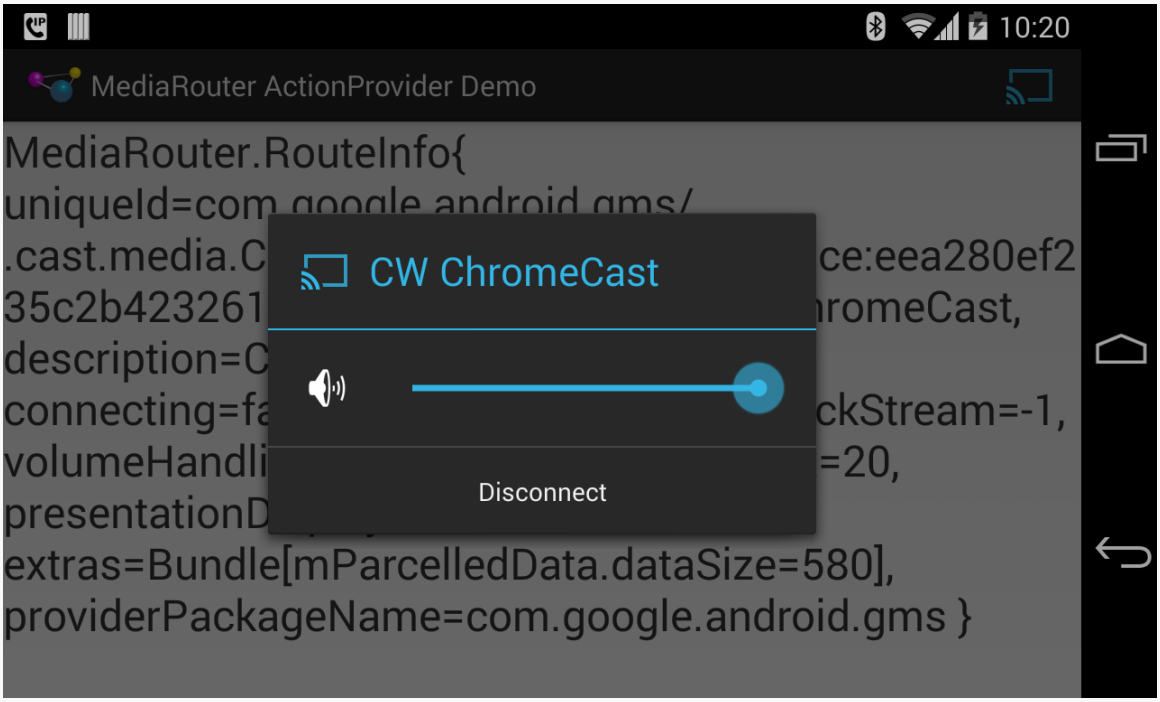


Figure 830: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Route Dialog

Tapping that "Disconnect" button returns everything to its original state.

Using Live Video Routes

A live video route is designed to be used with `Presentation`, a class that enables you to render your own content on the external display, much like how you would render your own content in a `Dialog`.

The use of `Presentation` is covered in an upcoming chapter.

Using Remote Playback Routes

In principle, `RemotePlaybackClient` allows you to work with remote playback routes, to specify `Uri` values to play back.

In practice, not even Google's own sample code for `RemotePlaybackClient` works reliably, let alone as documented.

That being said, let's take a look at the MediaRouter/RemotePlayback sample project, to see how `RemotePlaybackClient` works and where the current problems lie.

Setting Up MediaRouteActionProvider

Much of the basic setup of this application mirrors the `MediaRouteActionProvider` sample shown earlier in this chapter. One difference is that the UI is now encapsulated in a `PlaybackFragment`, with `MainActivity` simply setting up that fragment when needed:

```
package com.commonsware.android.remoteplayback;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
      getSupportFragmentManager().beginTransaction()
                                 .add(android.R.id.content,
                                      new PlaybackFragment()).commit();
    }
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/MainActivity.java)

`PlaybackFragment`, when it is created, opts into being retained on configuration changes, tells Android that it wishes to add items to the action bar, and sets up a `MediaRouteSelector` for `CATEGORY_REMOTE_PLAYBACK` routes:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
    setHasOptionsMenu(true);
    selector=
        new MediaRouteSelector.Builder()
            .addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK).build();
  }
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

Then, in `onAttach()` — called when the `PlaybackFragment` is attached to the hosting activity — we obtain a `MediaRouter` instance:

```
  @Override
  public void onAttach(Activity host) {
```

```
    super.onAttach(host);

    router=MediaRouter.getInstance(host);
  }
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

In `onStart()`, we hook a `cb` data member — an instance of `MediaRouter.Callback` up to the `MediaRouter`, also requesting that the `MediaRouter` initiate discovery of available routes. We remove our callback in `onStop()`:

```
@Override
public void onStart() {
  super.onStart();

  router.addCallback(selector, cb,
                     MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);
}

@Override
public void onStop() {
  router.removeCallback(cb);

  super.onStop();
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

We will examine `cb`'s declaration later in this section.

Later on, as part of our `onCreateOptionsMenu()` processing, we configure the `MediaRouteActionProvider` as before:

```
    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

    provider.setRouteSelector(selector);
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

All of this is very similar to the earlier examples. From here, though, we will actually use the route once the user selects it, to play back some media.

The Rest of the User Interface

The UI of the `PlaybackFragment` — other than the action bar — consists of a "transcript". This is a `TextView` inside of a `ScrollView`:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/transcript"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="20sp"/>

</ScrollView>
```

(from MediaRouter/RemotePlayback/app/src/main/res/layout/activity_main.xml)

As with most fragments, we inflate this layout in `onCreateView()`, holding onto the `TextView` and `ScrollView` widgets:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    scroll=
        (ScrollView)inflater.inflate(R.layout.activity_main, container,
                                     false);

    transcript=(TextView)scroll.findViewById(R.id.transcript);

    logToTranscript("Started");

    return(scroll);
  }
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

The `logToTranscript()` method will append a `String` to the `TextView` contents on a new line, plus scroll to the bottom to ensure that the new text is visible:

```
  private void logToTranscript(String msg) {
    if (client != null) {
      String sessionId=client.getSessionId();
```

```
    if (sessionId != null) {
      msg="(" + sessionId + ") " + msg;
    }
  }

  transcript.setText(transcript.getText().toString() + msg + "\n");
  scroll.fullScroll(View.FOCUS_DOWN);
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

The `client` data member referred to in `logToTranscript()` is our `RemotePlaybackClient` instance, which will be covered in the next section.

What the user sees when first running the sample is the action bar (with our `MediaRouteActionProvider`) and the transcript, with a simple "Started" message:

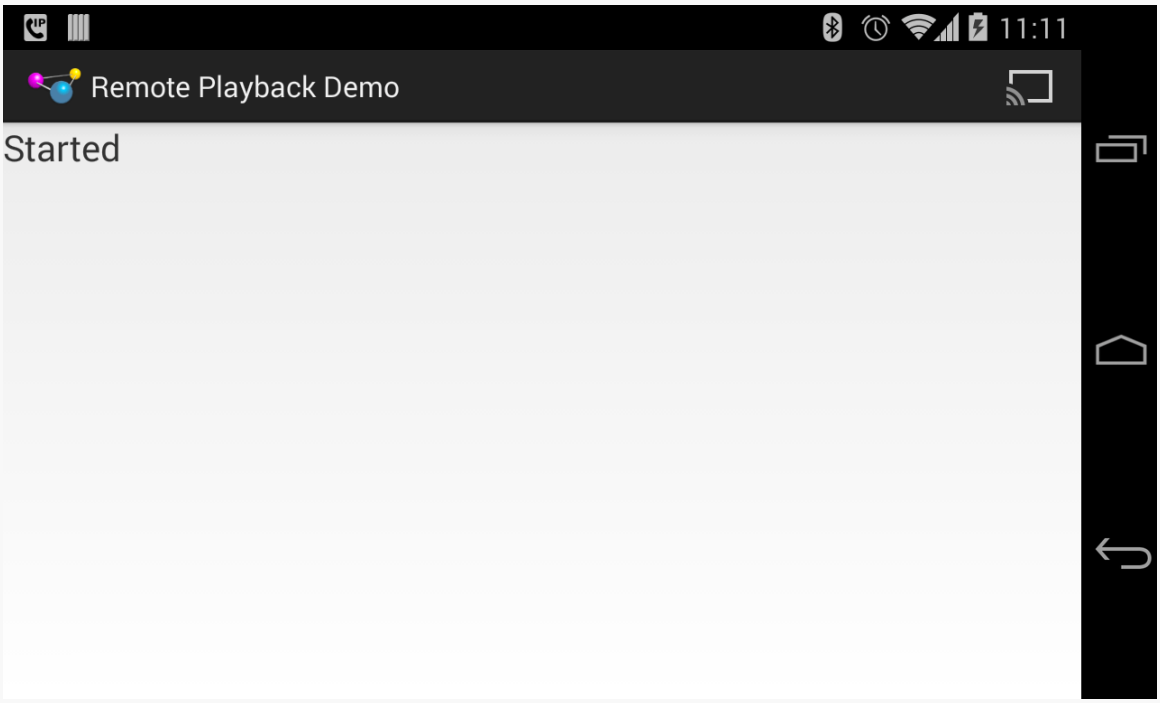


Figure 831: RemotePlaybackClient Demo, on a Nexus 4, As Initially Launched

As before, tapping on the "cast" action bar item pops up our dialog of available routes:

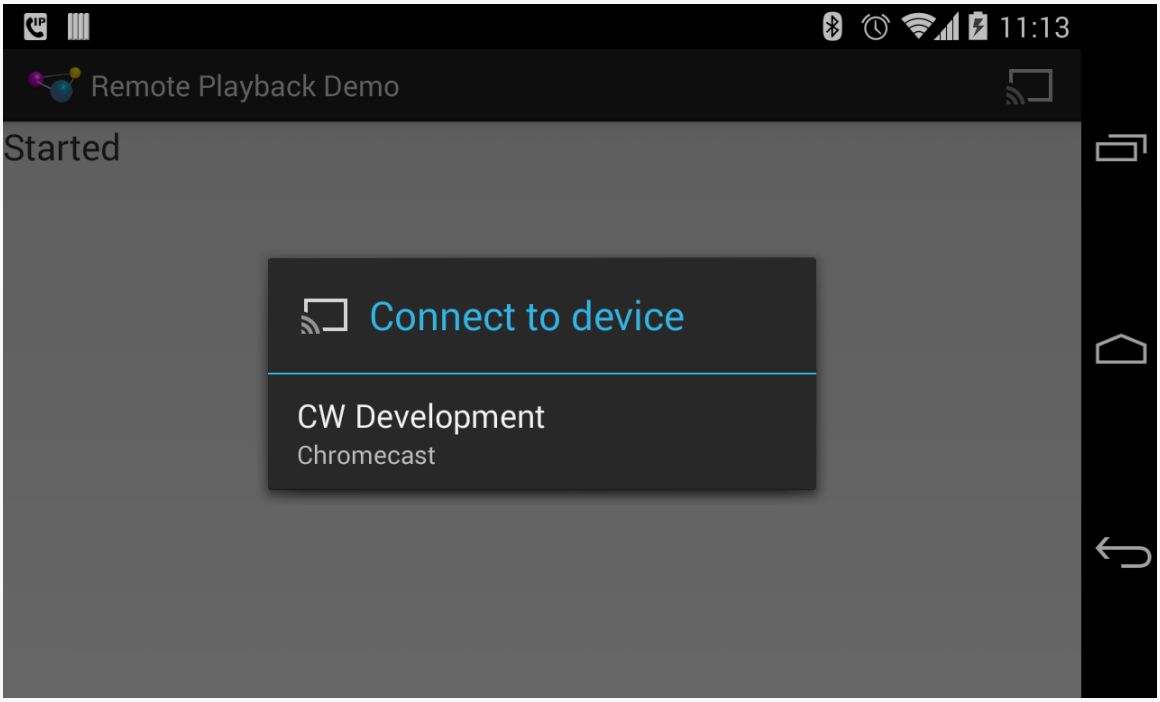


Figure 832: RemotePlaybackClient Demo, on a Nexus 4, Showing Available Routes

Connecting and Session Management

When the user selects a route, our `MediaRouter.Callback` (`cb`) is called with `onRouteSelected()`. Similarly, if the user elects to disconnect via the `MediaRouteActionProvider`, our `Callback` is called with `onRouteUnselected()`. In the `MediaRouter.Callback` implementation inside `PlaybackFragment`, those events route to `connect()` and `disconnect()` methods, respectively, after logging a message to the transcript:

```
private MediaRouter.Callback cb=new MediaRouter.Callback() {
  @Override
  public void onRouteSelected(MediaRouter router,
                              MediaRouter.RouteInfo route) {
    logToTranscript(getActivity().getString(R.string.route_selected));
    connect(route);
  }

  @Override
  public void onRouteUnselected(MediaRouter router,
                                MediaRouter.RouteInfo route) {
    logToTranscript(getActivity().getString(R.string.route_unselected));
    disconnect();
```

```
    }
};
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

The `connect()` method handles connecting to the remote playback device and starting a session:

```
private void connect(MediaRouter.RouteInfo route) {
  client=
      new RemotePlaybackClient(getActivity().getApplication(), route);

  if (client.isRemotePlaybackSupported()) {
    logToTranscript(getActivity().getString(R.string.connected));

    if (client.isSessionManagementSupported()) {
      client.startSession(null, new SessionActionCallback() {
        @Override
        public void onResult(Bundle data, String sessionId,
                             MediaSessionStatus sessionStatus) {
          logToTranscript(getActivity().getString(R.string.session_started));
          updateMenu();
        }

        @Override
        public void onError(String error, int code, Bundle data) {
          logToTranscript(getActivity().getString(R.string.session_failed));
        }
      });
    }
    else {
      getActivity().supportInvalidateOptionsMenu();
    }
  }
  else {
    logToTranscript(getActivity().getString(R.string.remote_playback_not_supported));
    client=null;
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

All of that, though, requires a bit more explanation.

What's a Session?

The objective of the `connect()` method is to establish a "session" with the `RemotePlaybackClient`. In Android's terms, a "session" is the state associated with an application's interactions with the remote playback client. In principle, the session could be shared among several instances of the app, such as several people contributing tracks to a dynamic playlist for audio playback at a party. Here, though, we are simply focused on having this one application instance have a session.

In principle, not all remote playback clients may support session management. In those cases, everybody is considered to be part of the same session. The test device for this sample (Chromecast) does support session management, however.

Connecting the Client

Connecting to the remote playback device is simply a matter of creating an instance of `RemotePlaybackClient`, specifying the route to connect to:

```
    client=
        new RemotePlaybackClient(getActivity().getApplication(), route);
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

Here, we use `getActivity().getApplication()` in the `RemotePlaybackClient` constructor. That is because we want to hold onto this `RemotePlaybackClient` instance across configuration changes, so we can easily maintain our session. Since we do not know what `RemotePlaybackClient` may hold onto given the supplied `Context`, and since we do not want to leak our activity by retaining a reference to it, we use the global `Application` instance, for a "leak-resistant" `Context`.

We also call `isRemotePlaybackSupported()` to confirm that, indeed, the `RemotePlaybackClient` is connected to something that supports remote playback. This should always return `true` in this case, as we are only interested in remote playback routes. But, a little defensive programming never hurts.

Assuming that is all OK, we log a "connected" message to the transcript and continue on to start our session.

Starting a Session

`isSessionManagementSupported()` on `RemotePlaybackClient` will indicate if the device supports explicit session management or not. If not, we will use the default implicit session and just continue on.

Otherwise, we call `startSession()` to explicitly start a session. This takes an optional `Bundle` of additional information to send in the start-session request to the device (or `null` if unused), plus a `SessionActionCallback`. The `SessionActionCallback` is supposed to be called when the session is ready for use. Surprisingly enough, this actually works... for `startSession()`.

The `SessionActionCallback` will be called with `onResult()` for success and `onError()` for failure. In either case, we log a message to the transcript indicating the status.

In addition, if we have a session — either explicitly created via `startSession()` or implicitly created for devices without explicit session management — we call an `updateMenu()` method to update the action bar items.

About the Action Bar

The fragment maintains two `boolean` values representing key states in the operation of the playback:

1. `isPlaying` indicates if playback was started and not yet stopped
2. `isPaused` indicates if playback was paused and not yet resumed

The aforementioned `updateMenu()` implementation uses those, plus the existence of a non-`null` `client`, to configure the action bar items:

```
private void updateMenu() {
  if (menu != null) {
    menu.findItem(R.id.stop).setVisible(client != null && isPlaying);
    menu.findItem(R.id.pause).setVisible(client != null && isPlaying
                                         && !isPaused);
    menu.findItem(R.id.play)
        .setVisible(client != null && (!isPlaying || isPaused));
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

Specifically:

- When we are not playing, the `play` item is visible; when we *are* playing, the `stop` item is visible
- When we are not paused, the `pause` item is visible (`play` serves "double duty", handling starting playback from a stopped state and resuming playback from a paused state)

This is based on a cached copy of the `Menu` object, saved in `onCreateOptionsMenu()` as part of setting up the action bar:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  this.menu=menu;
  inflater.inflate(R.menu.main, menu);

  updateMenu();

  MenuItem item=menu.findItem(R.id.route_provider);
  MediaRouteActionProvider provider=
      (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

  provider.setRouteSelector(selector);
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

This is also where the logic shown previously for configuring the `MediaRouteActionProvider` resides.

Session IDs

A session has a `String` identifier. In principle, this can be shared with other instances of your application, to allow for shared management of the session.

In the case of this sample, the session ID is merely logged to the transcript for all messages that are tied to an active session.

Hence, when the user chooses a remote playback route from the `MediaRouteActionProvider`, the resulting UI should resemble:

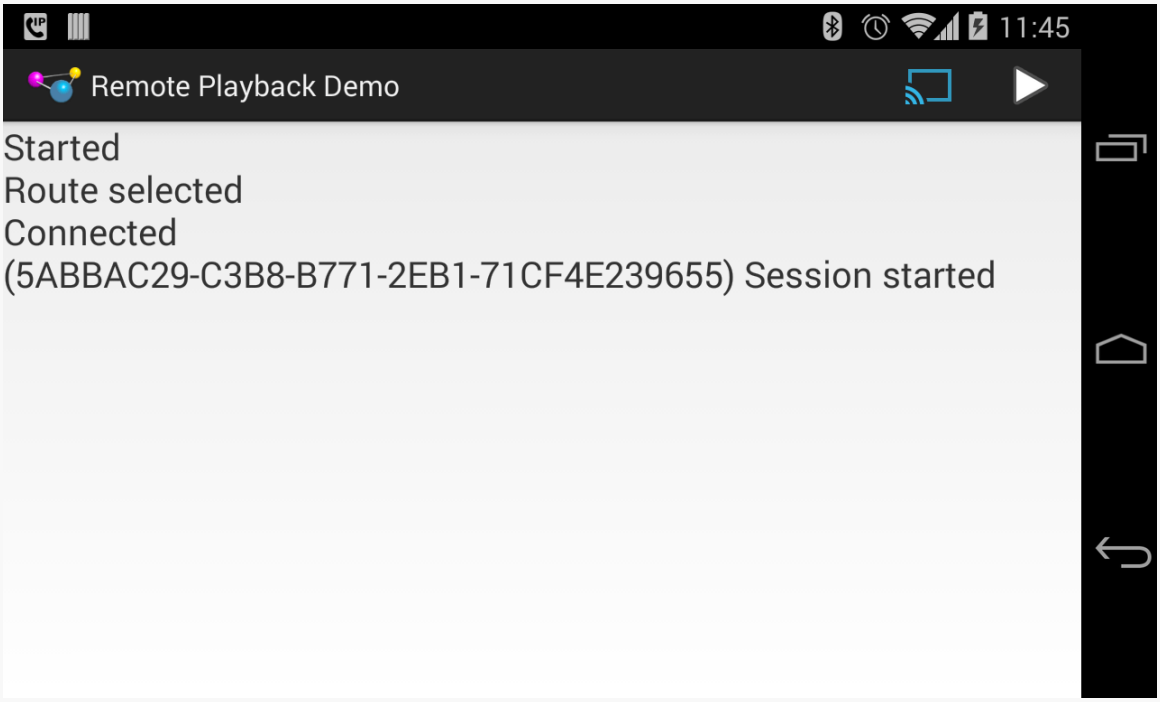


Figure 833: RemotePlaybackClient Demo, on a Nexus 4, Showing an Active Session

We see that we have connected to the client and started our session, and the `play` action bar item is now available to start playback of some media.

Playing

The `play` action bar item is tied to a `play()` method via `onOptionsItemSelected()`, if we are not paused:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.play:
      if (isPlaying && isPaused) {
        resume();
      }
      else {
        play();
      }

      return(true);

    case R.id.stop:
```

```
      stop();
      return(true);

    case R.id.pause:
      pause();
      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

`play()`, in turn, uses the `play()` method on `RemotePlaybackClient` to play back a copy of "Elephants Dream", a Creative Commons-licensed video, hosted on CommonsWare's corner of the Amazon S3 service:

```
private void play() {
  logToTranscript(getActivity().getString(R.string.play_requested));

  ItemActionCallback playCB=new ItemActionCallback() {
    @Override
    public void onResult(Bundle data, String sessionId,
                         MediaSessionStatus sessionStatus,
                         String itemId, MediaItemStatus itemStatus) {
      logToTranscript(getActivity().getString(R.string.playing));
      isPlaying=true;
      updateMenu();
    }

    @Override
    public void onError(String error, int code, Bundle data) {
      logToTranscript(getActivity().getString(R.string.play_error)
          + error);
    }
  };

  client.play(Uri.parse("http://misc.commonsware.com/ed_hd_512kb.mp4"),
              "video/mp4", null, 0, null, playCB);
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

The `play()` method on `RemotePlaybackClient` takes a few parameters:

- The `Uri` of the media to be played back
- The MIME type of that media (or `null` if you do not know the MIME type)

- An optional `Bundle` of metadata about the media to be played, where the `Bundle` keys come from `MediaItemMetadata` class (or `null` if none)
- The starting offset in the media to begin playback from (use `0` to start from the beginning)
- An optional `Bundle` of additional data to pass to the device
- An instance of `ItemActionCallback` to be notified when playback has started or has failed

`ItemActionCallback` is reminiscent of `SessionActionCallback`, in that `onResult()` will be called when playback begins and `onError()` will return when playback ends. The method signature of `onResult()` is slightly different, offering an ID and status of this particular media item.

In our case, we log a message to the transcript before requesting playback, then again on success or failure. On success, we also update `isPlaying` to be `true` and refresh the action bar.

Hence, once the user begins playback by tapping the `play` action bar item, the UI will look like this:

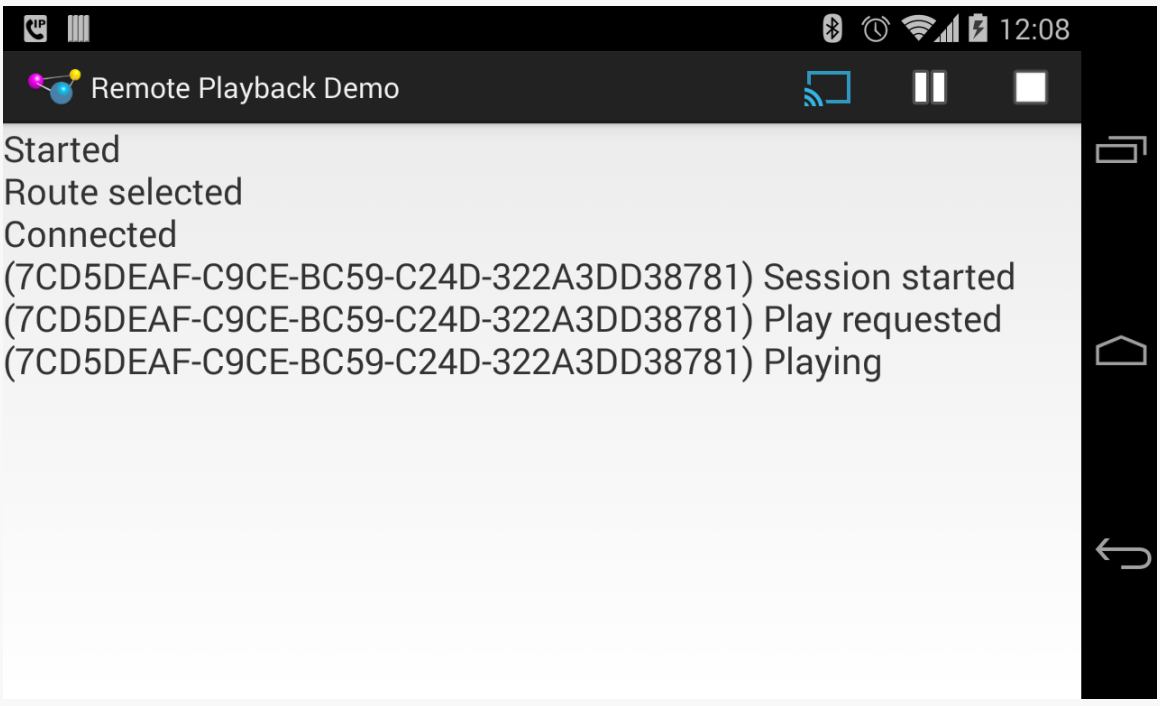


Figure 834: RemotePlaybackClient Demo, on a Nexus 4, After Playback Has Started

And, of course, the movie should be showing up on your remote playback device.

Stopping, and a Bug

The `stop()` action bar item is tied to a `stop()` method in `PlaybackFragment`. You would think that this would be very similar to starting playback — call some `stop()` method on `RemotePlaybackClient` and update the UI after playback has stopped.

And, indeed, that is what we do... except that we have to deal with a bug:

```
private void stop() {
  logToTranscript(getActivity().getString(R.string.stop_requested));

  StopCallback stopCB=new StopCallback();

  client.stop(null, stopCB);
  transcript.postDelayed(stopCB, 1000);
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

The stop() Call, and the Bug

`stop()` on `RemotePlaybackClient` takes an optional `Bundle` (here, `null`) and a `SessionActionCallback`. The `SessionActionCallback` is supposed to be called when playback has stopped (`onResult()`) or if there was some error in processing the request (`onError()`).

In practice, neither happen when testing this on a Chromecast. This same behavior can be seen with Google's own sample code, so it would not appear to be a problem with the author's own sample.

What actually happens is that playback is indeed stopped, but the `SessionActionCallback` is not called with `onResult()` or `onError()`.

The Workaround: RunnableSessionActionCallback

Since we cannot rely upon `onResult()` to be called for us, if we have work that we need to do in that case, we have to have some sort of fallback mechanism. One crude fallback is to assume that the request succeeded if we have not received a specific response after a period of time (say, 1000 milliseconds).

To that end, this sample has `RunnableSessionActionCallback`, a `SessionActionCallback` that implements `Runnable`:

```
abstract class RunnableSessionActionCallback extends
    SessionActionCallback implements Runnable {
  abstract protected void doWork();

  private boolean hasRun=false;

  @Override
  public void onResult(Bundle data, String sessionId,
                       MediaSessionStatus sessionStatus) {
    transcript.removeCallbacks(this);
    run();
  }

  @Override
  public void run() {
    if (!hasRun) {
      hasRun=true;
      doWork();
    }
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

The `run()` method sees whether or not the callback has already been run from a previous `run()` call. If not, it does the work specified by the `abstract` `doWork()` method, to be implemented in subclasses.

`StopCallback`, as seen in the `stop()` method above, extends `RunnableSessionActionCallback` and overrides `doWork()`:

```
private class StopCallback extends RunnableSessionActionCallback {
  @Override
  protected void doWork() {
    isPlaying=false;
    isPaused=false;
    updateMenu();
    logToTranscript(getActivity().getString(R.string.stopped));
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

`stop()` then not only passes the `StopCallback` to the `stop()` implementation on `RemotePlaybackClient`, but also schedules it as a `Runnable` to be invoked in 1000 milliseconds, via a call to `postDelayed()` on the `TextView` portion of the transcript. The `onResult()` implementation in `RunnableSessionActionCallback` calls `removeCallbacks()`, so we do not bother invoking the posted `Runnable` if that is not needed.

The `doWork()` implementation in `StopCallback` updates our flags, refreshes the action bar, and logs a message to the transcript. The result will look like:

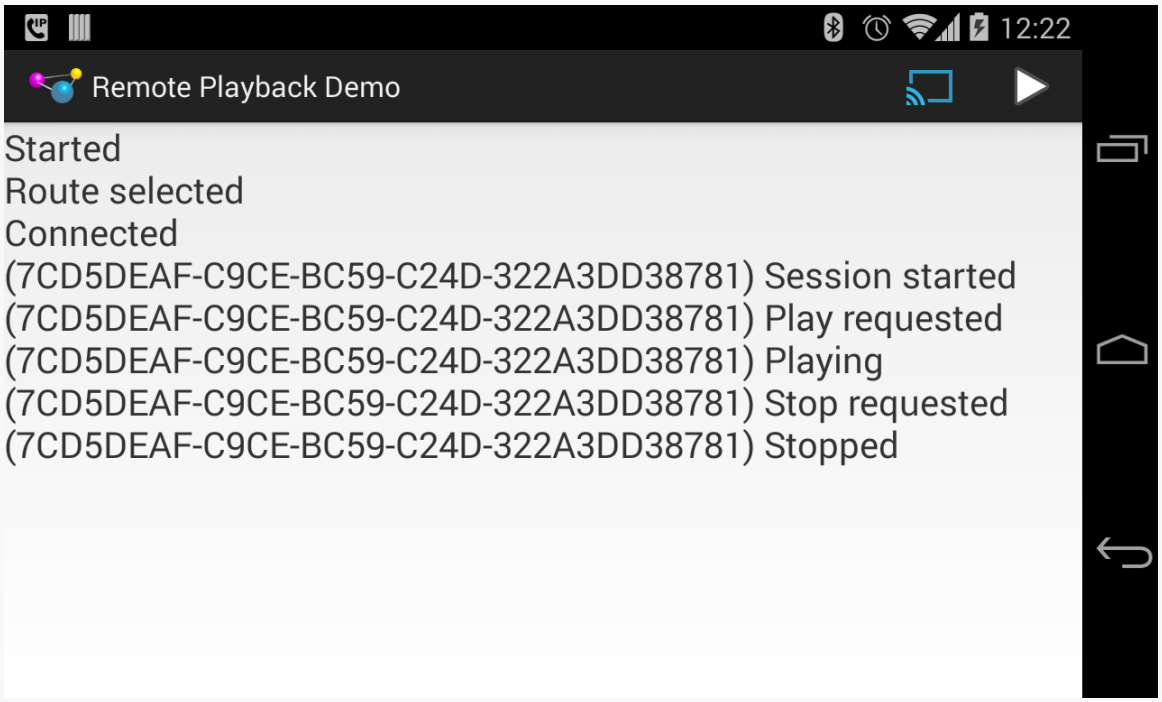


Figure 835: RemotePlaybackClient Demo, on a Nexus 4, After Playback Has Stopped

This sample also does not handle the case where the media completes playback on its own, insofar as this event is not detected, to update the action bar. This will be added in a future version of this sample, if further bugs allow such support to actually work.

Pausing and Resuming

Similarly, the `pause` action bar item forwards to a `pause()` method that calls `pause()` on the `RemotePlaybackClient`:

```
private void pause() {
  logToTranscript(getActivity().getString(R.string.pause_requested));

  PauseCallback pauseCB=new PauseCallback();

  client.pause(null, pauseCB);
  transcript.postDelayed(pauseCB, 1000);
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

That, in turn, uses `PauseCallback`:

```
private class PauseCallback extends RunnableSessionActionCallback {
  @Override
  protected void doWork() {
    isPaused=true;
    updateMenu();
    logToTranscript(getActivity().getString(R.string.paused));
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

This updates the action bar and logs messages to the transcript, similar to the `stop()` behavior. It also should successfully pause playback on the remote device.

The `play` action bar item routes to `resume()` if playback is paused:

```
private void resume() {
  logToTranscript(getActivity().getString(R.string.resume_requested));

  ResumeCallback resumeCB=new ResumeCallback();

  client.resume(null, resumeCB);
  transcript.postDelayed(resumeCB, 1000);
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

That, in turn, uses `ResumeCallback`:

```
private class ResumeCallback extends RunnableSessionActionCallback {
  @Override
  protected void doWork() {
    isPaused=false;
    updateMenu();
```

```
      logToTranscript(getActivity().getString(R.string.resumed));
    }
  }
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

This too updates the action bar and logs messages to the transcript, in addition to resuming playback on the remote device.

Disconnecting

A call to `disconnect()` on `PlaybackFragment` is triggered from two locations:

- `onRouteUnselected()` in our `MediaRouter.Callback`, such as when the user uses the `MediaRouteActionProvider` to disconnect from the route
- `onDestroy()`, as part of general cleanup of the fragment

`disconnect()` should reverse the work done in `connect()`, ending our session and releasing the client:

```
  private void disconnect() {
    isPlaying=false;
    isPaused=false;

    if (client != null) {
      logToTranscript(getActivity().getString(R.string.session_ending));
      EndSessionCallback endCB=new EndSessionCallback();

      if (client.isSessionManagementSupported()) {
        client.endSession(null, endCB);
      }

      transcript.postDelayed(endCB, 1000);
    }
  }
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

This simply calls the `endSession()` method on the `RemotePlaybackClient`, supplying an `EndSessionCallback` to be notified (theoretically) of when the session has been torn down. But it only calls `endSession()` if session management is supported; otherwise, we would get a runtime error.

To be sure we complete the disconnection, though, we schedule the `EndSessionCallback` as seen in the `stop()`, `pause()`, and `resume()` methods.

`EndSessionCallback` calls `release()` on the `RemotePlaybackClient`, to indicate that we are done with it, before setting `client` to `null`, refreshing the action bar, and logging something to the transcript:

```
private class EndSessionCallback extends
    RunnableSessionActionCallback {
  @Override
  protected void doWork() {
    client.release();
    client=null;

    if (getActivity() != null) {
      updateMenu();
      logToTranscript(getActivity().getString(R.string.session_ended));
    }
  }
}
```

(from MediaRouter/RemotePlayback/app/src/main/java/com/commonsware/android/remoteplayback/PlaybackFragment.java)

Other Remote Playback Features

There are other things that `RemotePlaybackClient` offers that are not shown in this sample:

- `enqueue()` allows you to build up a queue of media to be played back in the current session. This could be used by an individual or, in principle, by several people using the same app with a shared session ID. `remove()` allows you to remove specific items from the playback queue. These methods only work if `isQueueingSupported()` returns `true`.
- `getStatus()` will return information about the currently-playing piece of media, while `getSessionStatus()` will return information about the overall session. You can also find out about these changes on the fly by registering with `setStatusCallback()`.
- `seek()` allows you to move the playback to a new offset within the media, for "rewind" and "fast-forward" functionality. The status APIs (above) can tell you where you are in the playback, so you can determine the appropriate offset to seek to.

Supporting External Displays

Android 4.2 inaugurated support for applications to control what appears on an external or "secondary" display (e.g., TV connected via HDMI), replacing the default screen mirroring. This is largely handled through a `Presentation` object, where you declare the UI that goes onto the external display, in parallel with whatever your activity might be displaying on the primary screen.

In this chapter, we will review how Android supports these external displays, how you can find out if an external display is attached, and how you can use `Presentation` objects to control what is shown on that external display.

The author would like to thank Mark Allison, whose "Multiple Screens" blog post series helped to blaze the trail for everyone in this space.

Prerequisites

In addition to the core chapters, you should read the chapter on dialogs and the chapter on `MediaRouter` before reading this chapter.

A History of External Displays

In this chapter, "external displays" refers to a screen that is temporarily associated with an Android device, in contrast with a "primary screen" that is where the Android device normally presents its user interface. So, most Android devices connected to a television via HDMI would consider the television to be a "external display", with the touchscreen of the device itself as the "primary screen". However, a Android TV box or a Fire TV connected to a television via HDMI would consider the television to be the "primary screen", simply because there is no other screen. Some

devices themselves may have multiple screens, such as the Sony Tablet P — what those devices do with those screens will be up to the device.

Historically, support for external displays was manufacturer-dependent. Early Android devices had no ability to be displayed on an external display except through so-called "software projectors" like Jens Riboe's Droid@Screen. Some Android 2.x devices had ports that allowed for HDMI or composite connections to a television or projector. However, control for what would be displayed resided purely in the hands of the manufacturer. Some manufacturers would display whatever was on the touchscreen (a.k.a., "mirroring"). Some manufacturers would do that, but only for select apps, like a built-in video player.

Android 3.0 marked the beginning of Android's formal support for external displays, as the Motorola XOOM supported mirroring of the LCD's display via an micro-HDMI port. This mirroring was supplied by the core OS, not via device-dependent means. Any Android 3.0+ device with some sort of HDMI connection (e.g., micro-HDMI port) should support this same sort of mirroring capability.

However, mirroring was all that was possible. There was no means for an application to have something on the external display (e.g., a video) and something *else* on the primary screen (e.g., playback controls plus IMDB content about the movie being watched).

Android 4.2 changed that, with the introduction of `Presentation`.

What is a Presentation?

A `Presentation` is a container for displaying a UI, in the form of a `View` hierarchy (like that of an activity), on an external display.

You can think of a `Presentation` as being a bit like a `Dialog` in that regard. Just as a `Dialog` shows its UI separate from its associated activity, so does a `Presentation`. In fact, as it turns out, `Presentation` *inherits from* `Dialog`.

The biggest difference between a `Presentation` and an ordinary `Dialog`, of course, is where the UI is displayed. A `Presentation` displays on an external display; a `Dialog` displays on the primary screen, overlaying the activity. However, this difference has a profound implication: the *characteristics* of the external display, in terms of size and density, are likely to be different than those of a primary screen.

Hence, the resources used by the UI on an external display may be different than the resources used by the primary screen. As a result, **the `Context` of the `Presentation` is not the `Activity`**. Rather, it is a separate `Context`, one whose `Resources` object will use the proper resources based upon the external display characteristics.

This seemingly minor bit of bookkeeping has some rippling effects on setting up your `Presentation`, as we will see as this chapter unfolds.

Playing with External Displays

To write an app that uses an external display via a `Presentation`, you will need Android 4.2 or higher.

Beyond that, though, you will also need an external display of some form. Presently, you have three major options: emulate it, use a screen connected via some sort of cable, or use Miracast for wireless external displays (on Android 6.0 and older devices).

Emulated

Even without an actual external display, you can lightly test your `Presentation`-enabled app via the Developer Options area of Settings on your Android 4.2 device. There, in the Drawing category, you will see the "Simulate secondary displays" preference:

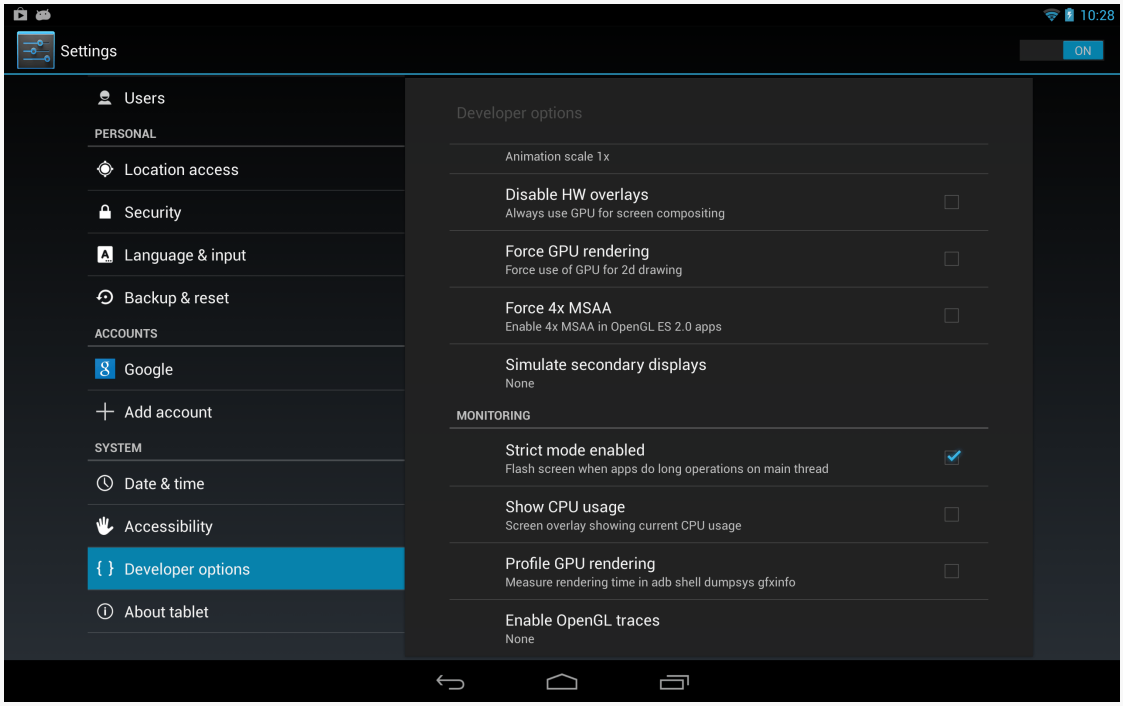


Figure 836: Nexus 10 "Simulate secondary displays" Preference

Tapping that will give you various options for what secondary display to emulate:

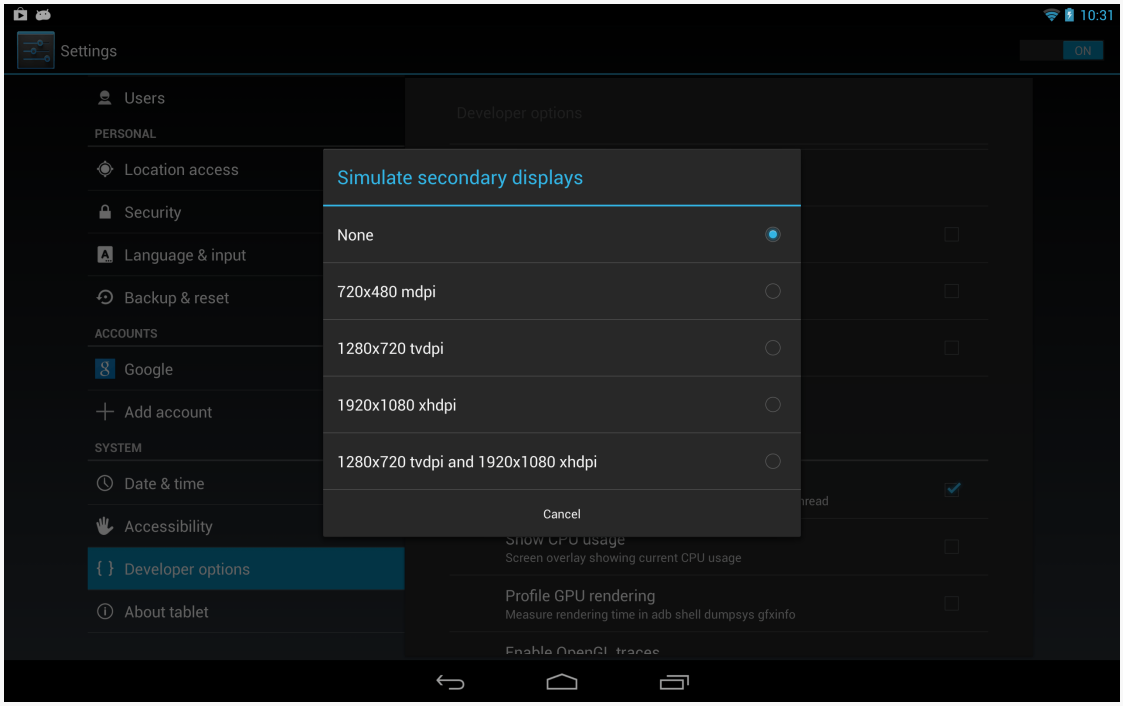


Figure 837: Nexus 10 "Simulate secondary displays" Options

Tapping one of those will give you a small window in the upper-left corner, showing the contents of the external display, overlaid on top of your regular screen:

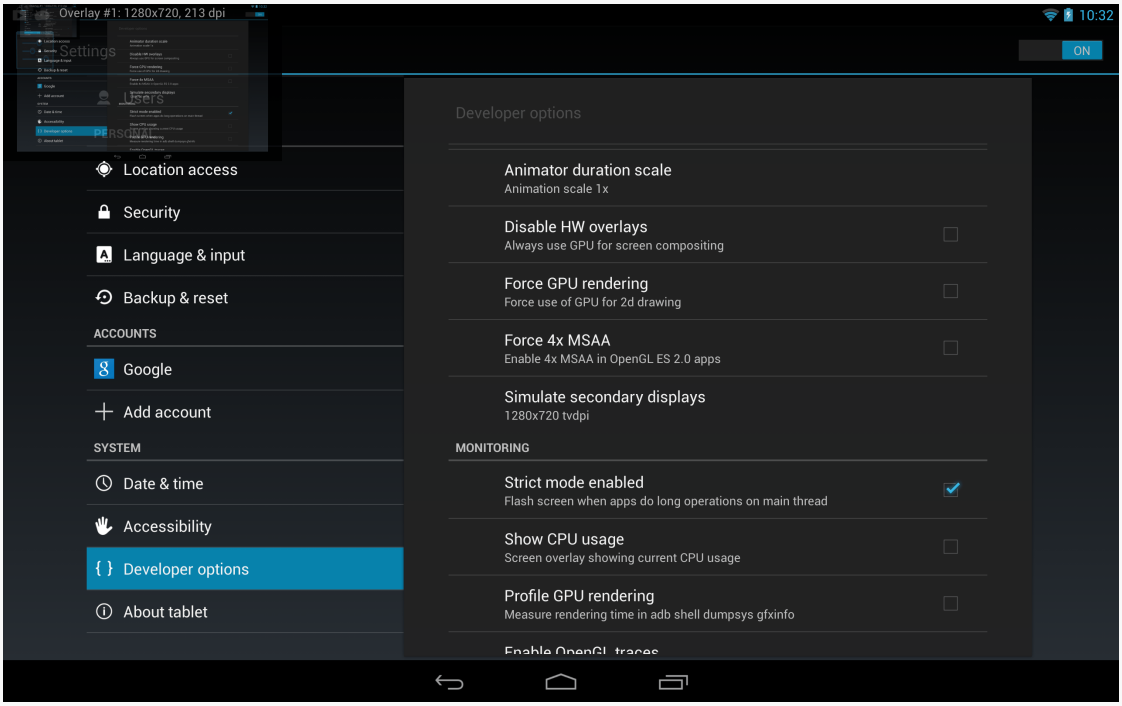


Figure 838: Nexus 10, Simulating a 720p external display

Normally, that will show a mirrored version of the primary screen, but with a `Presentation`-enabled app, it will show what is theoretically shown on the real external display.

However, there are limits with this technology:

- You will see this option on an Android emulator, but it may not work, particularly if you are not capable of using the "Host GPU Support" option. At the time of this writing, it works on the x86 Android 4.2 emulator image, but not the x86 Android 4.3 or 4.4 emulator image, and the ARM emulators are likely to be far too slow.
- The external display is rather tiny, making it difficult for you to accurately determine if everything is sized appropriately.
- The external display occludes part of the screen, overlaying your activities, though you can at least drag it around the screen to move it out of your way as needed.

In practice, before you ship a `Presentation`-capable app, you will want to test it with an actual physical external display.

HDMI

If you have a device with HDMI-out capability, and you have the appropriate cable, you can simply plug that cable between your device and the display. "Tuning" the display to use that specific HDMI input port should cause your device's screen contents to be mirrored to that display. Once this is working, you should be able to control the contents of that display using `Presentation`.

MHL

Mobile High-Definition Link, or MHL for short, is a relatively new option for connections to displays. On many modern Android devices, the micro USB port supports MHL as well. Some external displays have MHL ports, in which case a male-to-male MHL direct cable will connect the device to the display. Otherwise, MHL can be converted to HDMI via adapters, so an MHL-capable device can attach to any HDMI-compliant display.

SlimPort

SlimPort is another take on the overload-the-micro-USB-port-for-video approach. MHL is used on substantially more devices, but SlimPort appears on several of the Nexus-series devices (Nexus 4, Nexus 5, and the 2013 generation of the Nexus 7). Hence, while users will be more likely to have an MHL device, *developers* may be somewhat more likely to have a SlimPort device, given the popularity of Nexus devices among Android app developers.

From the standpoint of your programming work, MHL and SlimPort are largely equivalent — there is nothing that you need to do with your `Presentation` to address either of those protocols, let alone anything else like native HDMI.

USB 3.1 Type C

The new USB 3.1 Type C specification has enough hooks for video display that we may see Android devices starting to use it (along with USB->HDMI adapters) for supporting external displays.

Miracast

There are a few wireless display standards available. Android 4.2-6.0 supports Miracast, based upon WiFiDirect. This is also supported by some devices running

earlier versions of Android, such as some Samsung devices (where Miracast is sometimes referred to as "AllShare Cast"). However, unless and until those devices get upgraded to Android 4.2, you cannot control what they display, except perhaps through some manufacturer-specific APIs.

On a Miracast-capable device, going into Settings > Displays > Wireless display will give you the ability to toggle on wireless display support and scan for available displays:

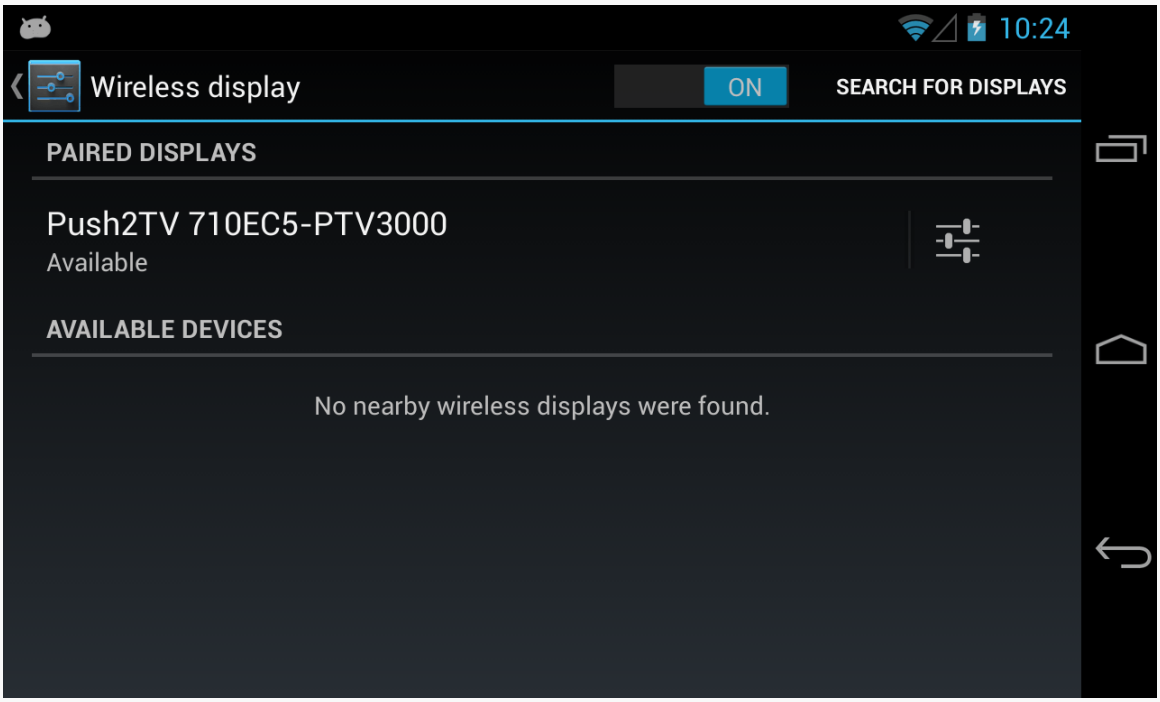


Figure 839: Nexus 4 Wireless Display Settings

You can then elect to attach to one of the available wireless displays and get your screen mirrored, and later use this with your `Presentation`-enabled app.

Of course, you also need some sort of Miracast-capable display, typically via some form of add-on box that connect to normal displays via HDMI and make them available via Miracast. One such box is the Netgear PTV3000, whose current firmware supports Miracast along with other wireless display protocols.

Note that Miracast uses a compressed protocol, to minimize the bandwidth needed to transmit the video. This, in turn, can cause some lag.

Note that Intel's WiDi is an extended version of Miracast.

WirelessHD

An up-and-coming competitor to Miracast is WirelessHD. WirelessHD has greater bandwidth requirements. On the other hand, it avoids compression, and therefore the lag that you experience with Miracast. At the time of this writing, though, no WirelessHD-native Android devices are available.

Detecting Displays

Of course, we can only present a `Presentation` on an external display if there is, indeed, such a screen available. There are two approaches for doing this: using `DisplayManager` and using `MediaRouter`. We examined `MediaRouter` for detecting live video routes in a preceding chapter, so let's focus here on `DisplayManager`.

`DisplayManager` is a system service, obtained by calling `getSystemService()` and asking for the `DISPLAY_SERVICE`.

Once you have a `DisplayManager`, you can ask it to give you a list of all available displays (`getDisplays()` with zero arguments) or all available displays in a certain category (`getDisplays()` with a single `String` parameter). As of API Level 17, the only available display category is `DISPLAY_CATEGORY_PRESENTATION`. The difference between the two flavors of `getDisplays()` is just the sort order:

- The zero-argument `getDisplays()` returns the `Display` array in arbitrary order
- The one-argument `getDisplays()` will put the `Display` objects matching the identified category earlier in the array

These would be useful if you wanted to pop up a list of available displays to ask the user which `Display` to use.

You can also register a `DisplayManager.DisplayListener` with the `DisplayManager` via `registerDisplayListener()`. This listener will be called when displays are added (e.g., HDMI cable was connected), removed (e.g., HDMI cable was disconnected), or changed. It is not completely clear what would trigger a "changed" call, though possibly an orientation-aware display might report back the revised height and width.

Note that while `DisplayManager` was added in API Level 17, `Display` itself has been around since API Level 1, though some additions have been made in more recent Android releases. But, this may mean that you can pass the `Display` object around to code supporting older devices without needing to constantly check for SDK level or add the `@TargetApi()` annotation.

Also note that the `support-v4` library contains a `DisplayManagerCompat`, allowing you to call `DisplayManager`-like methods going all the way back to API Level 4. This does not give older devices the ability to work with external displays — that would require a time machine — but it can make it incrementally easier for you to write your app, without having to worry about API level. `DisplayManagerCompat` just gracefully degrades to returning information only about the device's standard touchscreen.

A Simple Presentation

Let's take a look at a small sample app that demonstrates how we can display custom content on an external display using a `Presentation`. The app in question can be found in the Presentation/Simple sample project.

The Presentation Itself

Since `Presentation` extends from `Dialog`, we provide the UI to be displayed on the external display via a call to `setContentView()`, much like we would do in an activity. Here, we just create a `WebView` widget in Java, point it to some Web page, and use it:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private class SimplePresentation extends Presentation {
  SimplePresentation(Context ctxt, Display display) {
    super(ctxt, display);
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    WebView wv=new WebView(getContext());

    wv.loadUrl("https://commonsware.com");

    setContentView(wv);
```

```
    }
  }
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

However, there are two distinctive elements of our implementation:

- Our constructor takes a `Context` (typically the `Activity`), along with a `Display` object indicating where the UI should be presented.
- Our call to the `WebView` constructor uses `getContext()`, instead of the `Activity` object. In this case, that may have no real-world effect, as `WebView` is not going to be using any of our resources. But, had we used a `LayoutInflater` for inflating our UI, we would need to use one created from `getContext()`, not from the activity itself.

Detecting the Displays

We need to determine whether there is a suitable external display when our activity comes into the foreground. We also need to determine if an external display was added or removed while we are in the foreground.

So, in `onStart()`, if we are on an Android 4.2 or higher device, we will get connected to the `MediaRouter` to handle those chores:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
@Override
protected void onStart() {
  super.onStart();

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    if (cb==null) {
      cb=new RouteCallback();
      router=(MediaRouter)getSystemService(MEDIA_ROUTER_SERVICE);
    }

    handleRoute(router.getSelectedRoute(MediaRouter.ROUTE_TYPE_LIVE_VIDEO));
    router.addCallback(MediaRouter.ROUTE_TYPE_LIVE_VIDEO, cb);
  }
}
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

Specifically, we:

- Create an instance of `RouteCallback`, an inner class of our activity that extends `SimpleCallback`
- Use `getSystemService()` to obtain a `MediaRouter`
- Call a `handleRoute()` method on our activity that will update our UI based upon the current video route, obtained by calling `getSelectedRoute()` on the `MediaRouter`
- Register the `RouteCallback` object with the `MediaRouter` via `addCallback()`

The `RouteCallback` object simply overrides `onRoutePresentationDisplayChanged()`, which will be called whenever there is a change in what screens are available and considered to be the preferred modes for video. There, we just call that same `handleRoute()` method that we called in `onStart()`:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN)
private class RouteCallback extends SimpleCallback {
  @Override
  public void onRoutePresentationDisplayChanged(MediaRouter router,
                                                RouteInfo route) {
    handleRoute(route);
  }
}
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

Hence, our business logic for showing the presentation is isolated in one method, `handleRoute()`.

Our `onStop()` method will undo some of the work done by `onStart()`, notably removing our `RouteCallback`. We will examine that more closely in the next section.

Showing and Hiding the Presentation

Our `handleRoute()` method will be called with one of two parameter values:

- The `RouteInfo` of the active route we should use for displaying the `Presentation`
- `null`, indicating that there is no route for such content, other than the primary screen

If we are passed the `RouteInfo`, it may represent the route we are already using, or possibly it may represent a different route entirely.

We need to handle all of those cases, even if some (switching directly from one route to another) may not necessarily be readily testable.

Hence, our `handleRoute()` method does its best:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private void handleRoute(RouteInfo route) {
  if (route == null) {
    clearPreso();
  }
  else {
    Display display=route.getPresentationDisplay();

    if (route.isEnabled() && display != null) {
      if (preso == null) {
        showPreso(route);
        Log.d(getClass().getSimpleName(), "enabled route");
      }
      else if (preso.getDisplay().getDisplayId() != display.getDisplayId()) {
        clearPreso();
        showPreso(route);
        Log.d(getClass().getSimpleName(), "switched route");
      }
      else {
        // no-op: should already be set
      }
    }
    else {
      clearPreso();
      Log.d(getClass().getSimpleName(), "disabled route");
    }
  }
}
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

There are five possibilities handled by this method:

- If the route is `null`, then we should no longer be displaying the `Presentation`, so we call a `clearPreso()` method that will handle that
- If the route exists, but is disabled or is not giving us a `Display` object, we also assume that we should no longer be displaying the `Presentation`, so we call `clearPreso()`
- If the route exists and seems ready for use, and we are not already showing a `Presentation` (our `preso` data member is `null`), we need to show the `Presentation`, which we delegate to a `showPreso()` method

- If the route exists, seems ready for use, but we are already showing a `Presentation`, and the ID of the new `Display` is different than the ID of the `Display` our `Presentation` had been using, we use *both* `clearPreso()` and `showPreso()` to switch our `Presentation` to the new `Display`
- If the route exists, seems ready for use, but we are already showing a `Presentation` on this `Display`, we do nothing and wonder why `handleRoute()` got called

Showing the `Presentation` is merely a matter of creating an instance of our `SimplePresentation` and calling `show()` on it, like we would a regular `Dialog`:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private void showPreso(RouteInfo route) {
  preso=new SimplePresentation(this, route.getPresentationDisplay());
  preso.show();
}
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

Clearing the `Presentation` calls `dismiss()` on the `Presentation`, then sets the `preso` data member to `null` to indicate that we are not showing a `Presentation`:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private void clearPreso() {
  if (preso != null) {
    preso.dismiss();
    preso=null;
  }
}
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

Our `onPause()` uses `clearPreso()` and `removeCallback()` to unwind everything:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
@Override
protected void onStop() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    clearPreso();

    if (router != null) {
      router.removeCallback(cb);
    }
  }
```

```
    super.onStop();
  }
```

(from Presentation/Simple/app/src/main/java/com/commonsware/android/preso/simple/MainActivity.java)

The Results

If you run this with no external display, you will just see a plain `TextView` that is the UI for our primary screen:

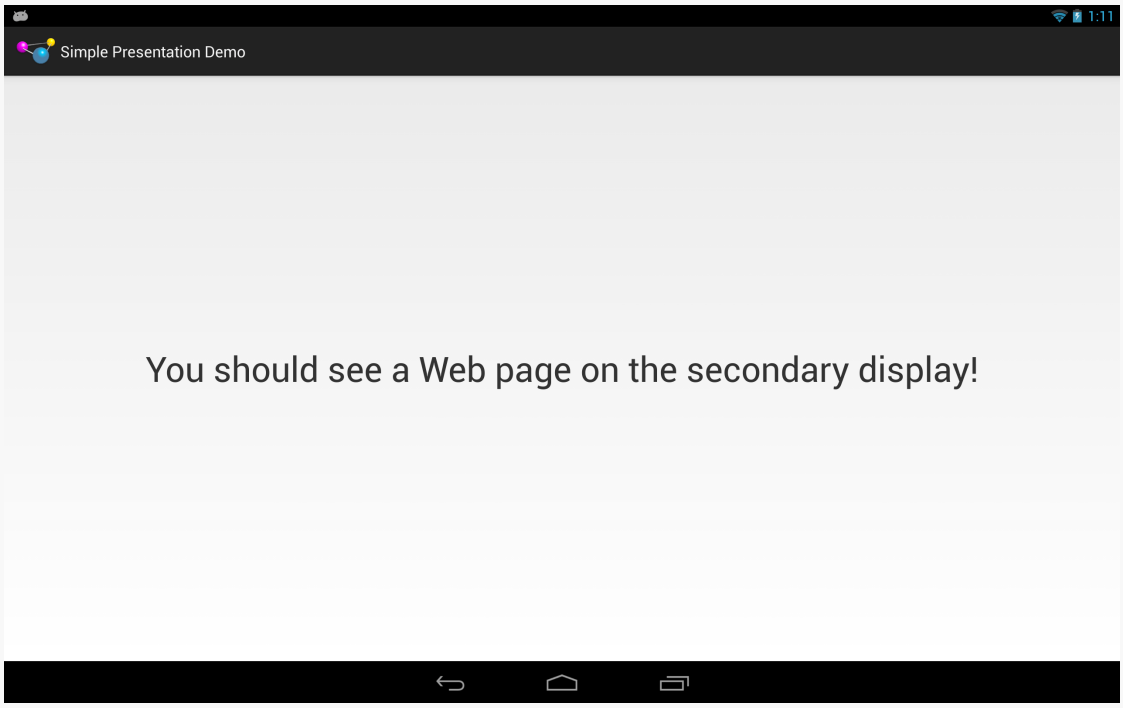


Figure 840: Nexus 10, No Emulated Secondary Display, Showing Sample App

If you run this *with* an external display, the external display will show our `WebView`:

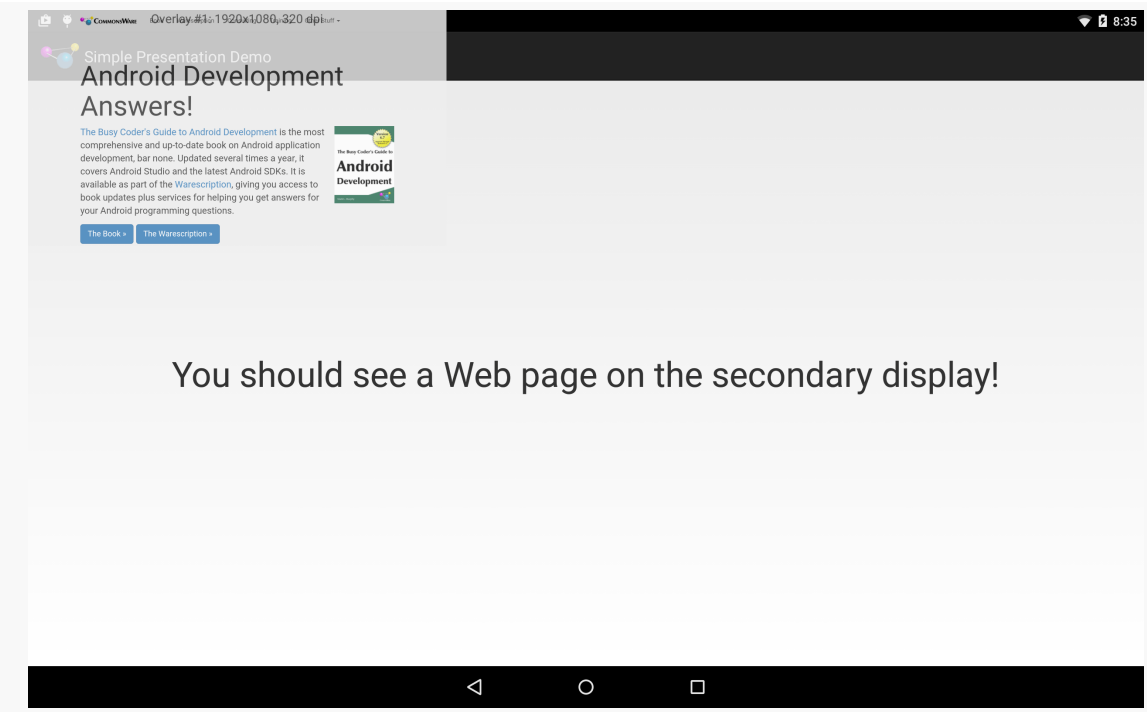


Figure 841: Nexus 10, With Emulated Secondary Display, Showing Sample App

A Simpler Presentation

There was a fair bit of code in the previous sample for messing around with `MediaRouter` and finding out about changes in the available displays.

To help simplify apps using `Presentation`, the author of this book maintains a library, `CWAC-Presentation`, with various reusable bits of code for managing `Presentations`.

One piece of this is `PresentationHelper`, which isolates all of the display management logic in a single reusable object. In this section, we will examine how to use `PresentationHelper`, then how `PresentationHelper` itself works, using `DisplayManager` under the covers.

Getting a Little Help

Our Presentation/Simpler sample project uses the CWAC-Presentation artifact:

```
apply plugin: 'com.android.application'

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'com.commonsware.cwac:presentation:0.4.+'
}

android {
    compileSdkVersion 19
    buildToolsVersion '26.0.2'
}
```

(from Presentation/Simpler/app/build.gradle)

This gives us access to `PresentationHelper`. Our `MainActivity` in the sample creates an instance of `PresentationHelper` in `onCreate()`, stashing the object in a data member:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);
    helper=new PresentationHelper(this, this);
  }
```

(from Presentation/Simpler/app/src/main/java/com/commonsware/android/preso/simpler/MainActivity.java)

The constructor for `PresentationHelper` takes two parameters:

- a `Context` object, one that should be valid for the life of the helper, typically the `Activity` that creates the helper, and
- a implementation of `PresentationHelper.Listener` — in this case, the interface is implemented on `MainActivity` itself

The activity that creates the helper must forward `onPause()` and `onResume()` lifecycle methods to the equivalent methods on the helper:

```
  @Override
  public void onResume() {
    super.onResume();
```

```
    helper.onResume();
  }

  @Override
  public void onPause() {
    helper.onPause();
    super.onPause();
  }
```

(from Presentation/Simpler/app/src/main/java/com/commonsware/android/preso/simpler/MainActivity.java)

The implementer of `PresentationHelper.Listener` also needs to have `showPreso()` and `clearPreso()` methods, much like the ones from the original `Presentation` sample in this chapter. `showPreso()` will be passed a `Display` object and should arrange to display a `Presentation` on that `Display`:

```
  @Override
  public void showPreso(Display display) {
    preso=new SimplerPresentation(this, display);
    preso.show();
  }
```

(from Presentation/Simpler/app/src/main/java/com/commonsware/android/preso/simpler/MainActivity.java)

`clearPreso()` should get rid of any outstanding `Presentation`. It is passed a `boolean` value, which will be `true` if we simply lost the `Display` we were using (and so the activity might want to display the `Presentation` contents elsewhere, such as in the activity itself), or `false` if the activity is moving to the background (triggered via `onPause()`):

```
  @Override
  public void clearPreso(boolean showInline) {
    if (preso != null) {
      preso.dismiss();
      preso=null;
    }
  }
```

(from Presentation/Simpler/app/src/main/java/com/commonsware/android/preso/simpler/MainActivity.java)

The implementations here are pretty much the same as the ones used in the previous example. `PresentationHelper` has handled all of the `Display`-management events – our activity can simply focus on showing or hiding the `Presentation` on demand.

Help When You Need It

In many respects, the `PresentationHelper` from the CWAC-Presentation project works a lot like the logic in the original `Presentation` sample's `MainActivity`, detecting various states and calling `showPreso()` and `clearPreso()` accordingly. However, `PresentationHelper` uses a different mechanism for this — `DisplayManager`.

The `PresentationHelper` constructor just stashes the parameters it is passed in data members and obtains a `DisplayManager` via `getSystemService()`, putting it in another data member:

```
/**
 * Basic constructor.
 *
 * @param ctxt a Context, typically the activity that is planning on showing
 *             the Presentation
 * @param listener the callback for show/hide events
 */
public PresentationHelper(Context ctxt, Listener listener) {
  this.listener=listener;

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    mgr=
        (DisplayManager)ctxt.getSystemService(Context.DISPLAY_SERVICE);
  }
}
```

`onResume()` calls out to a private `handlePreso()` method to initialize our state, and tells the `DisplayManager` to let it know as displays are attached and detached from the device, by means of `registerDisplayListener()`:

```
/**
 * Call this from onResume() of your activity, so we can determine what
 * changes need to be made to the Presentation, if any
 */
public void onResume() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    handleRoute();
    mgr.registerDisplayListener(this, null);
  }
}
```

The `PresentationHelper` itself implements the `DisplayListener` interface, which requires three callback methods:

- `onDisplayAdded()` is called when a new output display is available
- `onDisplayChanged()` is called when an existing attached display changes its characteristics
- `onDisplayRemoved()` is called whenever a previously-attached output display has been detached

In our case, all three methods route to the same `handleRoute()` method, to update our state:

```
/**
 * {@inheritDoc}
 */
@Override
public void onDisplayAdded(int displayId) {
  handleRoute();
}

/**
 * {@inheritDoc}
 */
@Override
public void onDisplayChanged(int displayId) {
  handleRoute();
}

/**
 * {@inheritDoc}
 */
@Override
public void onDisplayRemoved(int displayId) {
  handleRoute();
}
```

`handleRoute()` is where the bulk of the "business logic" of `PresentationHelper` resides:

```
private void handleRoute() {
  if (isEnabled()) {
    Display[] displays=
        mgr.getDisplays(DisplayManager.DISPLAY_CATEGORY_PRESENTATION);

    if (displays.length == 0) {
```

```
      if (current != null || isFirstRun) {
        listener.clearPreso(true);
        current=null;
      }
    }
    else {
      Display display=displays[0];

      if (display != null && display.isValid()) {
        if (current == null) {
          listener.showPreso(display);
          current=display;
        }
        else if (current.getDisplayId() != display.getDisplayId()) {
          listener.clearPreso(true);
          listener.showPreso(display);
          current=display;
        }
        else {
          // no-op: should already be set
        }
      }
      else if (current != null) {
        listener.clearPreso(true);
        current=null;
      }
    }

    isFirstRun=false;
  }
}
```

We get the list of attached displays from the `DisplayManager` by calling `getDisplays()`. By passing in `DISPLAY_CATEGORY_PRESENTATION`, we are asking for returned array of `Display` objects to be ordered such that the preferred display for presentations is the first element.

If the array is empty, and we already had a `current` `Display` from before (or if this is the first time `handlePreso()` has run), we call `clearPreso()` to inform the listener that there is no `Display` for presentation purposes.

If we do have a valid `Display`:

- If we were not displaying anything before, we call `showPreso()` to inform the listener to start displaying things, plus keep track of the `current` `Display` in a data member
- If we were displaying something before, but now the preferred `Display` for a `Presentation` is different (the ID value of the `Display` objects differ), we call `clearPreso()` and `showPreso()` to get the listener to switch to the new `Display`
- Otherwise, this was a spurious call to `handlePreso()`, so we do not do anything of note

If, for whatever reason, the best `Display` is not valid, we do the same thing as if we had no `Display` at all: call `clearPreso()`.

Finally, in `onPause()`, we call `clearPreso()` to ensure that we are no longer attempting to display anything, plus call `unregisterDisplayListener()` so we are no longer informed about changes to the mix of `Display` objects that might be available:

```
/**
 * Call this from onPause() of your activity, so we can determine what
 * changes need to be made to the Presentation, if any
 */
public void onPause() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    listener.clearPreso(false);
    current=null;

    mgr.unregisterDisplayListener(this);
  }
}
```

Presentations and Configuration Changes

One headache when using `Presentation` comes from the fact that it is a `Dialog`, which is owned by an `Activity`. If the device undergoes a configuration change, the activity will be destroyed and recreated by default, forcing you to destroy and recreate your `Dialog`. This, in turn, causes flicker on the external display, as the display briefly reverts to mirroring while this goes on.

Devices that support external displays may be orientation-locked to landscape when an external display is attached (e.g., an HDMI cable is plugged in). This reduces the odds of a configuration change considerably, as the #1 configuration change is an

orientation change. However, that is not a guaranteed "feature" of Android external display support, and there are other configuration changes that could go on (e.g., devices gets plugged into a keyboard dock).

You can either just live with the flicker, or use `android:configChanges` to try to avoid the destroy/re-create cycle for the configuration change. As was noted back in the chapter on configuration changes, this is a risky approach, as it requires you to remember all your resources that might change on the configuration change and reset them to reflect the configuration change.

A "middle ground" approach is to ensure that your activity running the `Presentation` is orientation-locked to landscape mode, by adding `android:orientation="landscape"` to your `<activity>` in the manifest, then use `android:configChanges` to handle the configuration changes related to orientation:

- `orientation`
- `keyboardHidden`
- `screenSize`
- `screenLayout`

For those configuration changes, nothing should be needed to be modified in your activity, since you want to be displaying in landscape all of the time, and so you will not need to modify your use of resources. This leaves open the possibility of other configuration changes that would cause flicker on the external display, but those are relatively unlikely to occur while your activity is in the foreground, and so it may not be worth trying to address the flicker in all those cases.

Yet another possibility is to have your presentation be delivered by a service, as we will discuss later in this chapter.

Presentations as Fragments

Curiously, the support for `Presentation` is focused on `View`. There is nothing built into Android 4.2 that ties a `Presentation` to a `Fragment`. However, this can be a useful technique, one we can roll ourselves... with a bit of difficulty.

The Reuse Reality

There will be a few apps that will only want to deliver content if there is a external display on which to deliver it. However, the vast majority of apps supporting external

displays will do so optionally, still supporting regular Android devices with only primary screens.

In this case, though, we have a problem: we need to show that UI *somewhere* if there is no external display to show it on. Our only likely answer is to have it be part of our primary UI.

Fragments would seem to be tailor-made for this. We could "throw" a fragment to the external display if it exists, or incorporate it into our main UI (e.g., as another page in a `ViewPager`) if the external display does not exist, or even have it be shown by some separate activity on smaller-screen devices like phones. Our business logic will already have been partitioned between the fragments — it is merely a question of where the fragment shows up.

Presentations as Dialogs

The nice thing is that `Presentation` extends `Dialog`. We already have a `DialogFragment` as part of Android that knows how to display a `Dialog` populated by a `Fragment` implementation of `onCreateView()`. `DialogFragment` even knows how to handle either being part of the main UI *or* as a separate dialog.

Hence, one could imagine a `PresentationFragment` that extends `DialogFragment` and adds the ability to either be part of the main UI on the primary screen *or* shown on an external display, should one be available.

And, in truth, it is possible to create such a `PresentationFragment`, though there are some limitations.

The Context Conundrum

The biggest limitation comes back to the `Context` used for our UI. Normally, there is only one `Context` of relevance: the `Activity`. In the case of `Presentation`, though, there is a separate `Context` that is tied to the display characteristics of the external display.

This means that `PresentationFragment` must manipulate *two* `Context` values:

- The `Activity`, if the fragment should be part of our main UI
- Some other `Context` supplied by the `Presentation`, if the fragment should be displayed in the `Presentation` on the external display

This makes creating a `PresentationFragment` class a bit tricky... though not impossible. After all, if it *were* impossible, these past several paragraphs would not be very useful.

A PresentationFragment (and Subclasses)

The Presentation/Fragment sample project has the same UI as the `Presentation/Simple` project, if there is an external display. If there is only the primary screen, though, we will elect to display the `WebView` side-by-side with our `TextView` in the main UI of our activity. And, to pull this off, we will create a `PresentationFragment` based on `DialogFragment`.

Note that this sample project has its `android:minSdkVersion` set to 17, mostly to cut down on all of the "only do this if we are on API Level 17" checks and `@TargetApi()` annotations. Getting this code to work on earlier versions of Android is left as an exercise for the reader.

In a simple `DialogFragment`, we might just override `onCreateView()` to provide the contents of the dialog. The default implementation of `onCreateDialog()` would create an empty `Dialog`, to be populated with the `View` returned by `onCreateView()`.

In our `PresentationFragment` subclass of `DialogFragment`, though, we need to override `onCreateDialog()` to use a `Presentation` instead of a `Dialog`... if we have a `Presentation` to work with:

```
package com.commonsware.android.preso.fragment;

import android.app.Dialog;
import android.app.DialogFragment;
import android.app.Presentation;
import android.content.Context;
import android.os.Bundle;
import android.view.Display;

abstract public class PresentationFragment extends DialogFragment {
  private Display display=null;
  private Presentation preso=null;

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    if (preso == null) {
      return(super.onCreateDialog(savedInstanceState));
    }
```

```
    return(preso);
  }

  public void setDisplay(Context ctxt, Display display) {
    if (display == null) {
      preso=null;
    }
    else {
      preso=new Presentation(ctxt, display, getTheme());
    }

    this.display=display;
  }

  public Display getDisplay() {
    return(display);
  }

  protected Context getContext() {
    if (preso != null) {
      return(preso.getContext());
    }

    return(getActivity());
  }
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/PresentationFragment.java)

We also expose `getDisplay()` and `setDisplay()` accessors, to supply the `Display` object to be used if this fragment will be thrown onto an external display. `setDisplay()` also creates the `Presentation` object wrapped around the display, using the three-parameter `Presentation` constructor that supplies the theme to be used (in this case, using the `getTheme()` method, which a subclass could override if desired).

`PresentationFragment` also implements a `getContext()` method. If this fragment will be used with a `Display` and `Presentation`, this will return the `Context` from the `Presentation`. If not, it returns the `Activity` associated with this `Fragment`.

This project contains a `WebPresentationFragment`, that pours the same basic Android source code used elsewhere in this book for a `WebViewFragment` into a subclass of `PresentationFragment`:

```
package com.commonsware.android.preso.fragment;
```

```
import android.annotation.TargetApi;
import android.os.Build;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.webkit.WebView;

public class WebPresentationFragment extends PresentationFragment {
  private WebView mWebView;
  private boolean mIsWebViewAvailable;

  /**
   * Called to instantiate the view. Creates and returns the
   * WebView.
   */
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    if (mWebView != null) {
      mWebView.destroy();
    }

    mWebView=new WebView(getContext());
    mIsWebViewAvailable=true;
    return mWebView;
  }

  /**
   * Called when the fragment is visible to the user and
   * actively running. Resumes the WebView.
   */
  @TargetApi(11)
  @Override
  public void onPause() {
    super.onPause();

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      mWebView.onPause();
    }
  }

  /**
   * Called when the fragment is no longer resumed. Pauses
   * the WebView.
   */
  @TargetApi(11)
```

```
  @Override
  public void onResume() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      mWebView.onResume();
    }

    super.onResume();
  }

  /**
   * Called when the WebView has been detached from the
   * fragment. The WebView is no longer available after this
   * time.
   */
  @Override
  public void onDestroyView() {
    mIsWebViewAvailable=false;
    super.onDestroyView();
  }

  /**
   * Called when the fragment is no longer in use. Destroys
   * the internal state of the WebView.
   */
  @Override
  public void onDestroy() {
    if (mWebView != null) {
      mWebView.destroy();
      mWebView=null;
    }
    super.onDestroy();
  }

  /**
   * Gets the WebView.
   */
  public WebView getWebView() {
    return mIsWebViewAvailable ? mWebView : null;
  }
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/WebPresentationFragment.java)

(note: the flawed comments came from the original Android open source code from which this fragment was derived)

The only significant difference, besides the superclass, is that the `onCreateView()` method uses `getContext()`, not `getActivity()`, as the `Context` to use when creating the `WebView`.

And, the project has a `SamplePresentationFragment` subclass of `WebPresentationFragment`, where we use the factory-method-and-arguments pattern to pass a URL into the fragment to use for populating the `WebView`:

```
package com.commonsware.android.preso.fragment;

import android.content.Context;
import android.os.Bundle;
import android.view.Display;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class SamplePresentationFragment extends WebPresentationFragment {
  private static final String ARG_URL="url";

  public static SamplePresentationFragment newInstance(Context ctxt,
                                                       Display display,
                                                       String url) {
    SamplePresentationFragment frag=new SamplePresentationFragment();

    frag.setDisplay(ctxt, display);

    Bundle b=new Bundle();

    b.putString(ARG_URL, url);
    frag.setArguments(b);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
        super.onCreateView(inflater, container, savedInstanceState);

    getWebView().loadUrl(getArguments().getString(ARG_URL));

    return(result);
```

```
  }
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/SamplePresentationFragment.java)

Using PresentationFragment

Our activity's layout now contains not only a `TextView`, but also a `FrameLayout` into which we will slot the `PresentationFragment` if there is no external display:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal"
  tools:context=".MainActivity">

  <TextView
    android:id="@+id/prose"
    android:layout_width="0px"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_weight="1"
    android:gravity="center"
    android:text="@string/secondary"
    android:textSize="40sp"/>

  <FrameLayout
    android:id="@+id/preso"
    android:layout_width="0px"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:visibility="gone"/>

</LinearLayout>
```

(from Presentation/Fragment/app/src/main/res/layout/activity_main.xml)

Note that the `FrameLayout` is initially set to have `gone` as its `visibility`, meaning that only the `TextView` will appear. Based on the widths and weights, the `TextView` will take up the full screen when the `FrameLayout` is `gone`, or they will split the screen in half otherwise.

In the `onCreate()` implementation of our activity (`MainActivity`), we inflate that layout and grab both the `TextView` and the `FrameLayout`, putting them into data members:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setContentView(R.layout.activity_main);

  inline=findViewById(R.id.preso);
  prose=(TextView)findViewById(R.id.prose);
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/MainActivity.java)

Our `onStart()` method, and our `RouteCallback`, are identical to those from the previous sample. Our `handleRoute()` method is *nearly* identical to the original, as is our `onStop()` method. The difference is that we need to distinguish whether we have lost an external display (and therefore want to move the Web page into the main UI) or if we are going away entirely (and therefore just wish to clean up the external display, if any). Hence, `clearPreso()` takes a `boolean` parameter (`switchToInline`), `true` if we want to show the fragment in the main UI, `false` otherwise. And, our `onStop()` and `handleRoute()` methods pass the appropriate value to `clearPreso()`:

```
@Override
protected void onStop() {
  clearPreso(false);

  if (router != null) {
    router.removeCallback(cb);
  }

  super.onStop();
}

private void handleRoute(RouteInfo route) {
  if (route == null) {
    clearPreso(true);
  }
  else {
    Display display=route.getPresentationDisplay();

    if (route.isEnabled() && display != null) {
      if (preso == null) {
        showPreso(route);
        Log.d(getClass().getSimpleName(), "enabled route");
      }
      else if (preso.getDisplay().getDisplayId() != display.getDisplayId()) {
        clearPreso(true);
```

```
        showPreso(route);
        Log.d(getClass().getSimpleName(), "switched route");
      }
      else {
        // no-op: should already be set
      }
    }
    else {
      clearPreso(true);
      Log.d(getClass().getSimpleName(), "disabled route");
    }
  }
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/MainActivity.java)

`showPreso()` is called when we want to display the `Presentation` on the external display. Hence, we need to remove the `WebPresentationFragment` from the main UI if it is there:

```
private void showPreso(RouteInfo route) {
  if (inline.getVisibility() == View.VISIBLE) {
    inline.setVisibility(View.GONE);
    prose.setText(R.string.secondary);

    Fragment f=getFragmentManager().findFragmentById(R.id.preso);

    getFragmentManager().beginTransaction().remove(f).commit();
  }

  preso=buildPreso(route.getPresentationDisplay());
  preso.show(getFragmentManager(), "preso");
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/MainActivity.java)

Creating the actual `PresentationFragment` is delegated to a `buildPreso()` method, which employs the `newInstance()` method on the `SamplePresentationFragment`:

```
private PresentationFragment buildPreso(Display display) {
  return(SamplePresentationFragment.newInstance(this, display,
                                                "https://commonsware.com"));
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/MainActivity.java)

`clearPreso()` is responsible for *adding* the `PresentationFragment` to the main UI, if `switchToInline` is `true`:

```
private void clearPreso(boolean switchToInline) {
  if (switchToInline) {
    inline.setVisibility(View.VISIBLE);
    prose.setText(R.string.primary);
    getFragmentManager().beginTransaction()
                        .add(R.id.preso, buildPreso(null)).commit();
  }

  if (preso != null) {
    preso.dismiss();
    preso=null;
  }
}
```

(from Presentation/Fragment/app/src/main/java/com/commonsware/android/preso/fragment/MainActivity.java)

With an external display, the results are visually identical to the original sample. Without an external display, though, our UI is presented side-by-side:

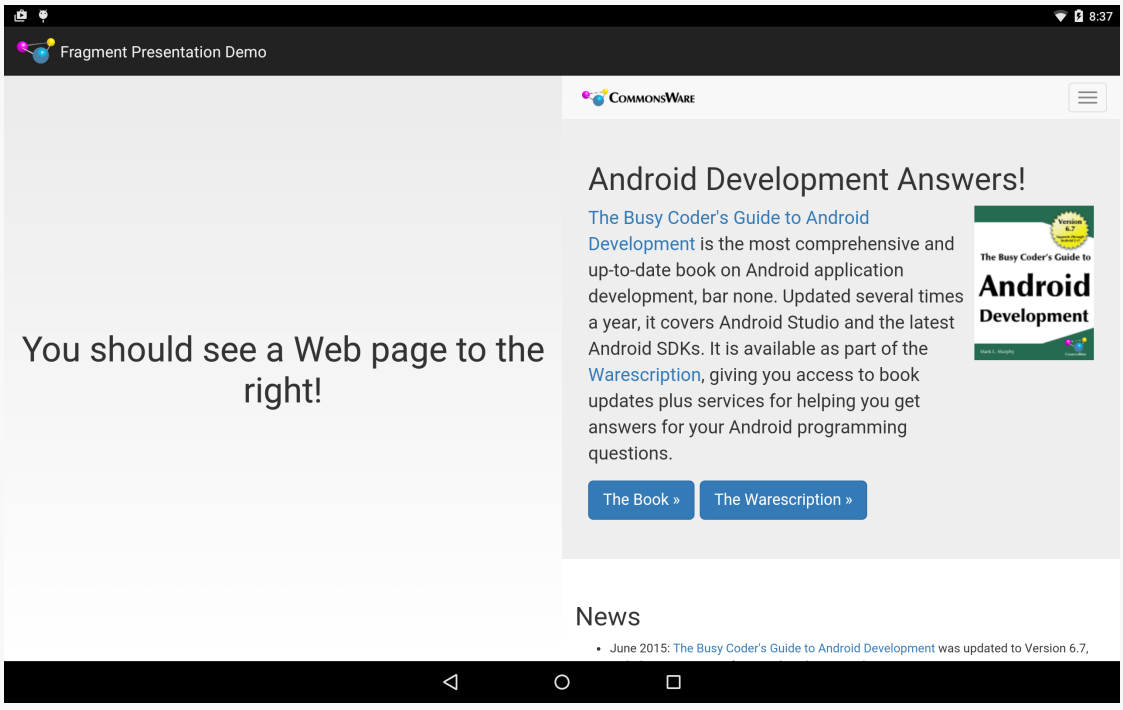


Figure 842: Nexus 10, With Inline PresentationFragment

Limits

This implementation of `PresentationFragment` has its limitations, though.

First, we cannot reuse the same fragment *instance* for both the inline UI and the `Presentation` UI, as they use different `Context` objects. Hence, production code will need to arrange to get data out of the old fragment instance and into the new instance when the screen mix changes. You might be able to leverage `onSaveInstanceState()` for that purpose, with a more-sophisticated implementation of `PresentationFragment`.

Also, depending upon the device and the external display, you *may* see multiple calls to `handleRoute()`. For example, attaching an external display may trigger three calls to your `RouteCallback`, for an attach, a detach, and another attach event. It is unclear why this occurs. However, it may require some additional logic in your app to deal with these events, if you encounter them.

Another Sample Project: Slides

At the 2013 Samsung Developer Conference, the author of this book delivered a presentation on using `Presentation`. Rather than use a traditional presentation package driven from a notebook, the author used the `Presentation/Slides` sample app. This sample app shows how to show slides on an external display, controlled by a `ViewPager` on a device's touchscreen.

What the audience saw, through most of the presentation, were simple slides. What the presenter saw was a `ViewPager`, with tabs, along with action bar items for various actions:

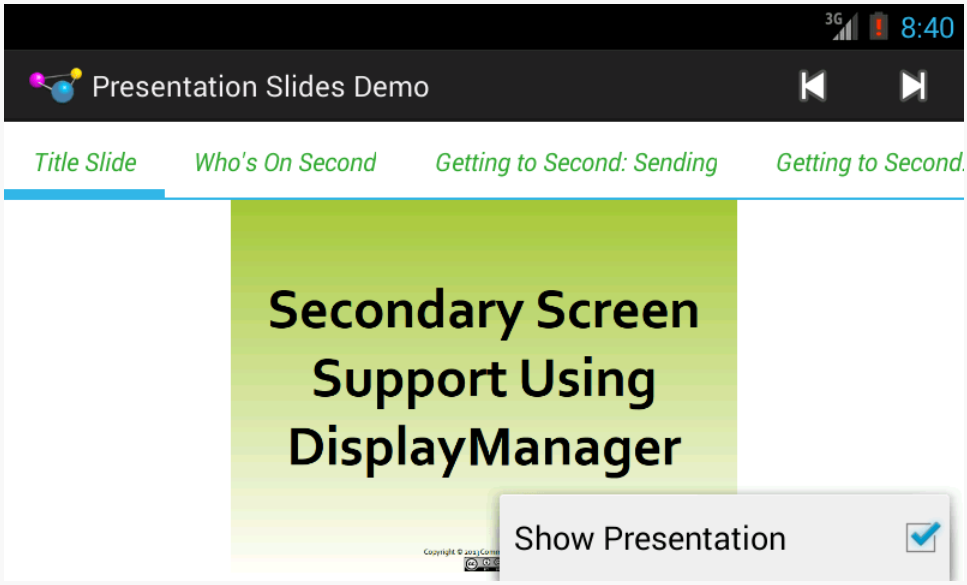


Figure 843: PresentationSlidesDemo, Showing Overflow

The Slides

The slides themselves are a series of 20 drawable resources (`img0`, `img1`, etc.), put into the `res/drawable-nodpi/` resource directory, as there is no intrinsic "density" that the slides were prepared for. As we use the slides in `ImageView` widgets, their images will be resized to fit the available `ImageView` space alone, not taking screen density into account.

There is a matching set of 20 string resources (`title0`, `title1`, etc.) containing a string representation of the slide titles, for use with `getPageTitle()` of a `PagerAdapter`.

The PagerAdapter

That `PagerAdapter`, named `SlidesAdapter`, has each slide be visually represented by an `ImageView` widget. In this case, `SlidesAdapter` extends `PagerAdapter` directly, skipping fragments:

```
package com.commonsware.android.preso.slides;

import android.content.Context;
import android.support.v4.view.PagerAdapter;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
```

```
class SlidesAdapter extends PagerAdapter {
  private static final int[] SLIDES= { R.drawable.img0,
      R.drawable.img1, R.drawable.img2, R.drawable.img3,
      R.drawable.img4, R.drawable.img5, R.drawable.img6,
      R.drawable.img7, R.drawable.img8, R.drawable.img9,
      R.drawable.img10, R.drawable.img11, R.drawable.img12,
      R.drawable.img13, R.drawable.img14, R.drawable.img15,
      R.drawable.img16, R.drawable.img17, R.drawable.img18,
      R.drawable.img19 };
  private static final int[] TITLES= { R.string.title0,
      R.string.title1, R.string.title2, R.string.title3,
      R.string.title4, R.string.title5, R.string.title6,
      R.string.title7, R.string.title8, R.string.title9,
      R.string.title10, R.string.title11, R.string.title12,
      R.string.title13, R.string.title14, R.string.title15,
      R.string.title16, R.string.title17, R.string.title18,
      R.string.title19 };
  private Context ctxt=null;

  SlidesAdapter(Context ctxt) {
    this.ctxt=ctxt;
  }

  @Override
  public Object instantiateItem(ViewGroup container, int position) {
    ImageView page=new ImageView(ctxt);

    page.setImageResource(getPageResource(position));
    container.addView(page,
                      new ViewGroup.LayoutParams(
                                              ViewGroup.LayoutParams.MATCH_PARENT,
                                              ViewGroup.LayoutParams.MATCH_PARENT));

    return(page);
  }

  @Override
  public void destroyItem(ViewGroup container, int position,
                          Object object) {
    container.removeView((View)object);
  }

  @Override
  public int getCount() {
    return(SLIDES.length);
  }

  @Override
  public boolean isViewFromObject(View view, Object object) {
    return(view == object);
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(TITLES[position]));
  }

  int getPageResource(int position) {
    return(SLIDES[position]);
```

```
  }
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/SlidesAdapter.java)

The data for the `SlidesAdapter` consists of a pair of static `int` arrays, one holding the drawable resource IDs, one holding the string resource IDs.

Of note, `SlidesAdapter` has a `getPageResource()` method, to return the drawable resource ID for a given page position, which is used by `instantiateItem()` for populating the position's `ImageView`.

The PresentationFragment

We also want to be able to show the slide on an external display via a `Presentation`. As with the preceding sample app, this one uses a `PresentationFragment`, here named `SlidePresentationFragment`:

```
package com.commonsware.android.preso.slides;

import android.content.Context;
import android.os.Bundle;
import android.view.Display;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import com.commonsware.cwac.preso.PresentationFragment;

public class SlidePresentationFragment extends PresentationFragment {
  private static final String KEY_RESOURCE="r";
  private ImageView slide=null;

  public static SlidePresentationFragment newInstance(Context ctxt,
                                                      Display display,
                                                      int initialResource) {
    SlidePresentationFragment frag=new SlidePresentationFragment();

    frag.setDisplay(ctxt, display);

    Bundle b=new Bundle();

    b.putInt(KEY_RESOURCE, initialResource);
    frag.setArguments(b);

    return(frag);
```

```
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    slide=new ImageView(getContext());

    setSlideContent(getArguments().getInt(KEY_RESOURCE));

    return(slide);
  }

  void setSlideContent(int resourceId) {
    slide.setImageResource(resourceId);
  }
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/SlidePresentationFragment.java)

Here, in addition to the sort of logic seen in the preceding sample app, we also need to teach the fragment which image it should be showing at any point in time. We do this in two ways:

1. We pass in an `int` named `initialResource` to the factory method, where `initialResource` represents the image to show when the fragment is first displayed. That value is packaged into the arguments `Bundle`, and `onCreateView()` uses that value.
2. Actually putting the drawable resource into the `ImageView` for this `Presentation` is handled by `setSlideContent()`. This is called by `onCreateView()`, passing in the `initialResource` value.

The Activity

The rest of the business logic for this application can be found in its overall entry point, `MainActivity`.

Setting Up the Pager

`onCreate()` of `MainActivity` is mostly focused on setting up the `ViewPager`:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);

    TabPageIndicator tabs=(TabPageIndicator)findViewById(R.id.titles);

    pager=(ViewPager)findViewById(R.id.pager);
    adapter=new SlidesAdapter(this);
    pager.setAdapter(adapter);
    tabs.setViewPager(pager);
    tabs.setOnPageChangeListener(this);

    helper=new PresentationHelper(this, this);
  }
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

The `ViewPager` and our `SampleAdapter` are saved in data members of the activity, for later reference. We also wire in a `TabPageIndicator`, from the ViewPagerIndicator library, and arrange to get control in our `OnPageChangeListener` methods when the page changes (whether via the tabs or via a swipe on the `ViewPager` itself).

`onCreate()` also hooks up a `PresentationHelper`, following the recipe used elsewhere in this chapter. And, as `PresentationHelper` requires, we forward along the `onResume()` and `onPause()` events to it:

```
@Override
public void onResume() {
  super.onResume();
  helper.onResume();
}

@Override
public void onPause() {
  helper.onPause();
  super.onPause();
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

Setting Up the Presentation

In the `showPreso()` method, required by the `PresentationHelper.Listener` interface, we create an instance of `SlidePresentationFragment`, passing in the resource ID of the current slide, as determined by the `ViewPager`:

```
@Override
public void showPreso(Display display) {
  int drawable=adapter.getPageResource(pager.getCurrentItem());

  preso=
      SlidePresentationFragment.newInstance(this, display, drawable);
  preso.show(getFragmentManager(), "preso");
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

We then `show()` the `PresentationFragment`, causing it to appear on the attached `Display`.

The corresponding `clearPreso()` method follows the typical recipe of calling `dismiss()` on the `PresentationFragment`, if one exists:

```
@Override
public void clearPreso(boolean showInline) {
  if (preso != null) {
    preso.dismiss();
    preso=null;
  }
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

Controlling the Presentation

However, the `SlidesPresentationFragment` now is showing the slide that was current when the `Display` was discovered or attached. What happens if the user changes the slide, using the `ViewPager`?

In that case, our `OnPageChangeListener` `onPageSelected()` method will be called, and we can update the `SlidesPresentationFragment` to show the new slide:

```
@Override
public void onPageSelected(int position) {
  if (preso != null) {
    preso.setSlideContent(adapter.getPageResource(position));
  }
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

Offering an Action Bar

The activity also sets up the action bar with three items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/first"
    android:icon="@android:drawable/ic_media_previous"
    android:showAsAction="always"
    android:title="@string/first">
  </item>
  <item
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:showAsAction="always"
    android:title="@string/last">
  </item>
    <item
    android:id="@+id/present"
    android:checkable="true"
    android:checked="true"
    android:showAsAction="never"
    android:title="@string/show_presentation">
  </item>

</menu>
```

(from Presentation/Slides/app/src/main/res/menu/activity_actions.xml)

Two, `first` and `last`, simply set the `ViewPager` position to be the first or last slide, respectively. This will also update the `SlidesPresentationFragment`, as `onPageSelected()` is called when we call `setCurrentItem()` on the `ViewPager`.

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.activity_actions, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.present:
        boolean original=item.isChecked();
```

```
      item.setChecked(!original);

      if (original) {
        helper.disable();
      }
      else {
        helper.enable();
      }

      break;

    case R.id.first:
      pager.setCurrentItem(0);
      break;

    case R.id.last:
      pager.setCurrentItem(adapter.getCount() - 1);
      break;
  }

  return(super.onOptionsItemSelected(item));
}
```

(from Presentation/Slides/app/src/main/java/com/commonsware/android/preso/slides/MainActivity.java)

The other action bar item, `present`, is a checkable action bar item, initially set to be checked. This item controls what we are showing on the external display:

- If it is checked, we want to show our `Presentation`
- If it is unchecked, we want to revert to default mirroring

The theory here is that, in a presentation, we could switch from showing the slides to showing the audience what the presenter has been seeing all along.

Switching between `Presentation` and default mirroring is a matter of calling `enable()` (to show a `Presentation`) or `disable()` (to revert to mirroring) on the `PresentationHelper`.

Device Support for Presentation

Alas, there is a problem: not all Android 4.2 devices support `Presentation`, even though they support displaying content on external displays. Non-`Presentation` devices simply support classic mirroring.

Generally speaking, it appears that devices that *shipped* with Android 4.2 and higher will support `Presentation`, assuming that they have some sort of external display support (e.g., MHL). Devices that were *upgraded* to Android 4.2 are less likely to support `Presentation`.

Unfortunately, at the present time, there is no known way to detect whether or not `Presentation` will work, let alone any means of filtering on this capability in the Play Store via `<uses-feature>`. With luck, this issue will be addressed in the future.

Presentations from a Service

Since `Presentation` inherits from `Dialog`, it also "inherits" one of the limitations of `Dialog`: you can only show one from an `Activity`. In many cases, that is not a big problem. If you are using the external display as an adjunct to your own app's use of the primary touchscreen, you would be using an activity anyway. However, it does prevent one from using `Presentation` to, say, implement a video player app that plays on an external display but does not tie up the touchscreen, so the user can use other apps while the video plays.

However, as it turns out, it *is* possible to drive the content of an external display from a background app... just not by using a `Presentation`.

The details of this are a bit tricky, derived from one Stack Overflow answer and another Stack Overflow question.

However, you do not need to deal with all of the details, courtesy of `PresentationService`.

`PresentationService` is a class in the CWAC-Presentation library. `PresentationService` clones some of the logic from `Presentation` and `Dialog`, enough to allow you to define a `View` that will be shown on external display, driven by a `Service`. `PresentationService` is an `abstract` base class for you to extend, where `PresentationService` handles showing your content on an external display, and you simply manage that content.

The CWAC-Presentation library has a `demoService/` directory containing a sample use of `PresentationService`. The recipe is fairly simple and is outlined in the following sections.

Step #1: Attach the Libraries

The CWAC-Presentation `README` contains instructions for attaching the libraries to your project, whether via Gradle dependencies, downloading a pair of JARs, or using the source form of the Android library project.

Step #2: Create a Stub PresentationService

As is noted above, `PresentationService` is an `abstract` class, so you will need to create your own concrete subclass of it, under whatever name you wish. And, as with any service, you will need a `<service>` element in the manifest. None of this is especially unusual.

The sample app is a service-based rendition of the `Presentation/Slides` sample app described earlier in this chapter. It has a `SlideshowService` that will display the slideshow on an external display from the background, switching slides every five seconds.

Step #3: Return the Theme

One of the `abstract` methods that you will need to implement is `getThemeId()`. This should return the value of the style resource that represents the theme that you wish to use for the widgets you are going to show on the external display.

For example, if your project uses the `@style/AppTheme` approach that is code-generated for you, you can simply return `R.style.AppTheme` from `getThemeId()`, as the sample app does:

```
  private int iteration=0;
  private Handler handler=null;

  @Override
```

Step #4: Build the View

The other `abstract` method you need to implement is `buildPresoView()`. You are passed a `Context` and a `LayoutInflater`, and your job is to use those to build your UI for the external display, returning the root view. The `LayoutInflater` is already set up to use the theme you provided via `getThemeId()`.

Since this will be called shortly before showing the result on the external display, you can also take this time to initialize other aspects of your presentation. For example, the `SlideshowService` implements `Runnable` and has a `Handler` for the main application thread, initialized in `onCreate()`:

```
      R.drawable.img1, R.drawable.img2, R.drawable.img3,
      R.drawable.img4, R.drawable.img5, R.drawable.img6,
      R.drawable.img7, R.drawable.img8, R.drawable.img9,
      R.drawable.img10, R.drawable.img11, R.drawable.img12,
      R.drawable.img13, R.drawable.img14, R.drawable.img15,
      R.drawable.img16, R.drawable.img17, R.drawable.img18,
      R.drawable.img19 };
```

`buildPresoView()` not only returns an `ImageView` for the slides, but also calls `run()`, which populates the `ImageView` and calls `postDelayed()` on the `Handler` to schedule `run()` to be called again in five seconds, thereby arranging to update the slide every five seconds:

```
    handler=new Handler(Looper.getMainLooper());
    super.onCreate();

    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    startForeground(1338, buildForegroundNotification());
  }
```

`onDestroy()` calls `removeCallbacks()` to break the `Handler` `postDelayed()` loop:

```
  public int onStartCommand(Intent intent, int flags, int startId) {
    if (ACTION_STOP.equals(intent.getAction())) {
      stopSelf();
    }

    return(super.onStartCommand(intent, flags, startId));
```

Step #5: Start and Stop the Service

Calling `startService()` on your service will then trigger the presentation. Or, more accurately, it will trigger `PresentationService` to work with a `PresentationHelper` to determine when a presentation should be shown. `PresentationService` will then use `buildPresoView()` to populate the external display. Conversely, calling `stopService()` will stop the presentation.

It is up to you to determine what is the trigger for these calls. The sample app simply starts the service immediately when run and stops the service in response to an action bar item click.

While the service is running, you are welcome to use an event bus or other means to control the contents of the presentation, by manipulating the widgets you created in `buildPresoView()`.

Note that it is safe to call `startService()` on the service multiple times, if you do not know whether the service is already running and need to ensure that it is running now.

Hey, What About Chromecast?

In February 2014, Google released a long-awaited SDK to allow anyone to write an app that connects to Chromecast, Google's streaming-media HDMI stick. This Cast SDK also works with other Google Cast-capable devices, like some Android TV models. A natural question coming out of that is whether `Presentation` and `DisplayManager` work with Chromecast.

The answer is: that depends on how you look at the problem.

While Chromecast may physically resemble a wireless display adapter, in truth it is its own device, running a customized mashup of Android and ChromeOS. Chromecast's strength is in playing streaming media from any source, primarily directly off of the Internet.

The classic approach for the Google Cast SDK is that apps are telling the Chromecast what to stream from, not streaming to the Chromecast itself. As such, the Cast API is distinctly different from that of `Presentation`, and while the two both deal with what the Android device would consider an external display, they are not equivalent solutions.

However:

- A Chromecast can also serve as a Miracast endpoint; if a user sets that up, then your app can use `Presentation` with a Chromecast
- In 2015, the Cast SDK added a `Presentation`-workalike API, one that presumably works with Chromecast without having to go through the Miracast setup

More coverage of Chromecast can be found in the next chapter.

Google Cast and Chromecast

A popular target for `MediaRouter`, in some countries, is Chromecast, Google's lightweight streaming media player for televisions and other HDMI displays. Originally, Chromecast was a "closed box", with no official support for third-party apps (and active work to block unofficial support). In early 2014, though, Google finally opened up Chromecast to developers.

This chapter covers what it takes to enable an Android app to "cast" content to a Chromecast, possibly as part of a broader external display strategy.

Prerequisites

In addition to the core chapters, you should read the chapter on `MediaRouter` before reading this chapter.

Here a Cast, There a Cast

You will see two terms used in this chapter and in the online literature regarding all of this: Chromecast and "Google Cast". Despite the similarities in their names, these are fairly distinct items.

What is Chromecast?

Chromecast, as noted earlier in this chapter, is a streaming media receiver, sold by Google under their own brand.

Figure 844: Google Chromecast

It plugs into an HDMI port of a television or similar display, plus uses micro USB for supplying power.

However, rather than other streaming media receivers, that use Bluetooth or IR (infrared) peripherals for controlling the playback, Chromecast appears to use WiFi, designed to be controlled by a smartphone, tablet, or Chrome Web browser.

Chromecast itself runs its own OS, apparently a hybrid of Android and ChromeOS.

What is Google Cast?

Google Cast can be thought of as a control protocol for Google Cast-enabled receivers. Through a Google-supplied SDK (or other means), Google Cast client apps ("senders") can direct a Google Cast-enabled receiver to play, pause, rewind, fast-forward, etc. a stream.

Google Cast could, in theory, be "baked into" displays (such as a television), in addition to being supported by dedicated media receivers like the Chromecast.

Google Cast does assume that, in general, the media receiver runs its own OS and is capable of playing streaming media without ongoing assistance from the Google Cast client. Hence, the client is not "locked into" having to keep feeding content to the Google Cast client, allowing the user to go off and do other things with that client while playback is going on.

Common Chromecast Development Notes

Chromecast goes to sleep if it detects that it is plugged into a television or monitor that is turned off (or perhaps even not accepting input from the HDMI port the Chromecast is using). While it is in this sleep mode, it may not appear as an available route. You may need to keep the display active to allow Chromecast to work properly. A 720p-capable pico projector, such as the Vivitek Qumi series, can be a handy way to have a test display for Chromecast (or for live video media routes) at your development station, without the bulk of another monitor, if you have a handy surface to project upon.

Also, note that a Chromecast "uses Google's DNS regardless of what you have defined locally", according to a Google engineer. That will preclude you from using any local domains on an organization's own DNS server, without some tricky firewall configuration to route Google DNS requests to the in-house DNS server. Similarly, you cannot use machine names as pseudo-domain names, the way you might be able to using a regular Web browser.

Your API Choices

Chromecast offers up remote playback media routes and works with `RemotePlaybackClient`, as is discussed in the chapter on `MediaRouter`. The sample app for `RemotePlaybackClient` was tested on a Chromecast.

If you want greater control than is offered via `RemotePlaybackClient`, though, you can use the Cast SDK. This SDK is part of the Play Services framework, not part of Android itself. It also works solely with Google Cast devices, of which Chromecast is the only known example, whereas other sorts of devices are able to publish remote playback media routes. Hence, using the Cast SDK will tie you to Google Cast — and some of its restrictions, both technical and legal — but will give you greater developer control over the behavior of both the Google Cast device and your app.

This chapter will focus on the Cast SDK. See the chapter on `MediaRouter` for coverage of `RemotePlaybackClient`.

Senders and Receivers

There are three major components to the Google Cast environment:

- The sources of streaming media, usually out on the Internet
- The software on the playback device that plays that streaming media ("receiver")
- The software on the control device (phone, tablet, Chrome Web browser) that directs the receiver about what to play and when ("sender")

The Sender App

The sender app is responsible for allowing the user to choose some media to play, then to control the actual playback (pause, start, stop, rewind, fast-forward, etc.).

The details of how to choose some media will depend heavily on the nature of the sender app. For example, a subscription-based streaming video service, such as Netflix, would allow the user to browse and search eligible content hosted by Netflix itself. Netflix presumably has its own Web service APIs that its own sender app would use for this purpose, and it is up to Netflix to offer a sensible UI for choosing a piece of media to watch.

Passing a reference (e.g., URL) to the receiver, and issuing control commands, will either be handled by `RemotePlaybackClient` (on Android) or via a Google-supplied SDK (for Android, iOS, or Chrome Web apps).

The Receiver

The details of how a receiver is implemented is up to the manufacturer of the Google Cast-enabled device. In the case of Chromecast, it is a version of the Chrome Web browser. In principle, the implementation could be anything; in practice, it is likely that the same basic software stack will be used, courtesy of licensing Google Cast technology from Google for streaming media devices.

Official Google Cast receiver software comes in three flavors: default, styled, and custom.

Default Receiver

The default receiver is what you get by default, as you might have guessed. If you do nothing else, your sender will be communicating with the default receiver. In effect, the default receiver is a specific Chrome Web app, running on the Chrome browser inside of the Chromecast, that is responsible for playback of your chosen media.

Other than providing the URLs to the media, plus requests to pause, start, stop, etc. the playback, you have no control over the default receiver, particularly from a look-and-feel standpoint.

Styled Receiver

A styled receiver is one where you, the developer, supply light branding information that is applied to what otherwise is the default receiver, such as a logo.

Whereas using the default receiver requires no explicit registration with Google, using the styled receiver does require you to register your sender app with Google, at which point you will be able to provide a URL pointing to a CSS file that contains the custom styles.

Custom Receiver

If you would rather replace the default receiver functionality with your own, either to offer more functionality, or to consume media types that may require additional configuration (e.g., DRM), you can create a custom receiver. This, in effect, is a Chrome Web app, where you provide not only CSS, but the HTML and JavaScript as well. This is substantially more complicated, and it requires registration with Google (as with the styled receiver). However, you have far greater control over what appears on the television.

Supported Media Types

The list of supported media types is likely to change over time. At present, Google Cast-enabled devices are supposed to support major media types, such as:

- MP4 and VP8 for video
- MP3, AAC, and Ogg Vorbis for audio
- PNG, JPEG, GIF, BMP, and WEBP for still images (e.g., photos)
- HLS and MPEG-DASH for streaming

Cast SDK Dependencies

Using the Cast SDK to develop for Google Cast devices has a fair number of dependencies... and not just dependencies on particular libraries.

Developer Registration

If you are going to be using the default receiver, *and* you do not need to have debugging access to the device (e.g., to examine JavaScript logs from the Web rendering engine on the Google Cast device), you are welcome to develop your apps independently.

However, if you will use a styled or custom receiver, or you wish to gain debugging access to the device, you will need to register with Google.

This process will involve you agreeing to some terms of service (see below), along with paying a $5 registration fee.

The Terms of Service

The Google Cast SDK has separate Developer Terms of Service from anything else. If you are going to use the Google Cast SDK, you will be expected to agree to these terms as part of the registration processes. You are strongly encouraged to review these terms with qualified legal counsel. Failure to comply with the terms may cause your app (or, more accurately, your styled or custom receivers) to "be de-registered", presumably meaning that it will no longer work.

These terms contain some curious clauses, worth discussing with your attorney, including a requirement to adhere to a massive design checklist, controlling the look-and-feel of your sender and receiver. This includes a specific requirement for the precise icon to be used for initiating communications with the Google Cast receiver. Those agreeing to these terms are also barred from doing things that might allow others to display content on a Google Cast receiver without using the SDK or breaking through any access controls on the Google Cast device (e.g., creating an exploit that roots it).

Device Registration and Development Setup

While registering your device is optional, it may be handy for custom receivers, so that you can debug your custom HTML and JavaScript that is being rendered by the Google Cast device.

First, you should configure your device to publish its serial number to Google when it checks for Google Cast software updates. For the Chromecast, this involves using whatever means you used to configure the Chromecast in the first place for your network (e.g., the Chromecast Android app). There should be an option for "Send this Chromecast's serial number when checking for updates" — in the Chromecast Android app, this will be in the "Share Data" section of the device's settings screen.

Once you have registered as a Cast SDK developer, the Google Cast SDK Developer Console will have an option for you to "Add New Device". You will need the Google Cast device's serial number — in the case of the Chromecast, this is etched on the underside of the device.

Note that it may take some time before your device registration will be complete, as the device will not find out about the registration until it checks for another update, and there does not appear to be a way to trigger this. Hence, you may need to wait a few hours. You will know that you have access once you can successfully connect, via a Web browser, to port 9222 on the IP address of the Google Cast device. For the Chromecast, the easiest way to get that IP address is through your Chromecast configuration tool (e.g., the Chromecast Android app). Note that the Web page may not be much (e.g., "Inspectable WebContents"), but it will not return a 404 or similar error code.

If you wish to use a styled or custom receiver, you will also need to register your application, in the same Cast SDK Console area. This will be covered in a future edition of this book.

The Official Libraries

You will need the Google Play Services SDK, which you may have used already for other portions of the Play Services framework, such as GCM, Maps V2, and so on.

You will also need the same `mediarouter` Android library project covered in the chapter on `MediaRouter`, along with its dependencies (e.g., the `support-v4` library and the `appcompat` library).

The CastCompanionLibrary… Or Not

The Play Services SDK (and its dependencies) is all that you need to write Cast SDK applications. However, Google has also published the Cast Companion Library (CCL), containing a lot of helper code to make it a bit easier for you to write apps that adhere to the design checklist

Developing Google Cast Apps

Coverage of the Cast SDK, including sample apps, will be added to this chapter in a future edition of this book.

The “Ten-Foot UI”

Increasingly, Android devices are being used to drive screens that are somewhat larger than those found on your average phone or tablet:

- Many Android phones and tablets can directly deliver content to TVs, monitors, and projectors via HDMI, MHL, SlimPort, Miracast, and similar technologies
- Android devices can control the behavior of non-Android presentation engines, like Chromecast
- Some Android devices themselves use a TV or other display as their primary screen, from big names (Google and Amazon), mid-range firms (OUYA), and firms you have never heard of (various Android “HDMI sticks” available on eBay, Alibaba, etc.)

Technically, writing for these displays is a bit different than you would do for a phone or tablet. In some cases, such as with Google Cast, writing for these displays is more substantially different.

However, in all cases, the design of the UI needs to be different, owing to different physical and usage characteristics of large screens. This chapter will focus on this so-called “ten-foot UI” and help you understand what sorts of changes will need to be considered.

Prerequisites

Understanding this chapter requires that you have read the chapter on focus management.

The sample of the "leanback" UI is a revised version of a sample app profiled in the chapter on the `MediaStore` `ContentProvider`.

What is the "Ten-Foot UI"?

The "ten-foot UI" is not referring to a UI that is 3.048 meters high or 9.87789527 by 10^-17 parsecs wide.

Rather, the distance referred to by the "ten-foot UI" indicates the approximate distance between the viewer and the screen. People usually sit farther from TVs, monitors, and projectors than they do phones or tablets when using them. Partly, that is because the screens are a lot bigger, so they do not need to sit as closely. Partly, that is because often times the screens are being "used" by more than one person (e.g., an audience watching a presentation on a projector), and everybody needs to be able to see the screen.

The expression "ten-foot UI" refers to the design constraints inherent in developing user interfaces to be used across such a distance. Even though the screen may be bigger, the *apparent* screen size (or "visual angle" may be no bigger than phones or tablets, or sometimes even less. That, plus user input differences, technical differences between TVs and other displays, and so on all go into the "ten-foot UI" design guidance that UI experts give us.

Overscan

Television standards have been with us for several decades. Television sets from the dawn of television had significantly lower and more variable quality than today's devices. The delivery of the signal at the outset had significantly lower and more variable quality than today's over-the-air HDTV or cable connections. As a result of these two characteristics, the engineers devising television standards made some decisions that, while necessary at the time, add some complexity to delivering apps to televisions, in the form of overscan.

Simply put, not all televisions show exactly the same picture. Depending on device and signal, a television may show up to 12% less of the picture, as measured horizontally and vertically. Hence, the theoretical ideal screen size (e.g., 720p = 1280 x 720 pixels) may be achieved in some cases, but you may get less (e.g., 1128 x 634 pixels) in other cases.

Android TV and Fire TV ignore overscan, relying upon developers to take it into account. As a result, the reported screen resolution is *not* necessarily available to you. Instead, you need to avoid putting anything important in the outer ~10% of the screen, centering the important stuff within the available space.

So, for example, you might have a background for your game (e.g., a starfield). Make sure that there is nothing essential on the background image that the user must see that is along the outside edge. Then, if part of the background is lost due to overscan, there is no particular problem.

The bigger issue, of course, is standard foreground widgets and containers. Android developers are used to being able to have layouts that work edge-to-edge, with just a minor amount of margin so text, icons, and the like do not run right into the edge of the screen. Now, you need more than a "minor amount" of margin. Google and Amazon recommend a `27dp` margin on the top and bottom sides of your activities, and a `48dp` margin on the left and right sides of your activities.

For activity and fragment layouts that are dedicated for TV presentation, you could elect to put those margins in those layouts, or add them via a theme. However, for activity and fragment layouts that may be used both for a touchscreen device (phone or tablet) and a television, adding the margin on the touchscreen device may be unsuitable.

For that, you could use dimension resources in different resource sets. Define `overscan_horizontal` and `overscan_vertical` to be both `0dp` (or whatever) in `res/values/dimens.xml`. Define them to be `48dp` and `27dp`, respectively, in `res/values-television/dimens.xml`, where `-television` is a resource set qualifier that will be used in Android TV and other TV-based Android devices. Then, you can refer to `@dimen/overscan_horizontal` and `@dimen/overscan_vertical` in activity/fragment layouts, to take overscan into account conditionally.

Navigation

Most televisions, monitors, and projectors are not touchscreens. Users will be changing what is shown either by using some sort of remote control (e.g., Fire TV, Android TV) or by using an app that runs on a touchscreen device (e.g., direct monitor connection, Chromecast).

In the remote control scenarios, in-screen navigation becomes important. Those remote controls usually focus on some sort of D-pad or arrow keys for moving focus

and clicking on widgets. This forces users into a sequential-access model (e.g., click "left" three times then "enter" once) rather than the random-access model that touchscreens offer.

The chapter on focus management covers these sorts of concerns. Bear in mind that getting focus management implemented properly in your app not only helps with the "ten-foot UI", but also can help other sorts of users, such as the visually impaired or motion impaired who cannot readily use touchscreens.

Also note that text input is a significant chore when you try to do it using a remote control. Hence, even to the extent it is possible, try to limit the number of places that users *have* to type into `EditText` widgets and the like in your UI. If possible, offer a way for users to do that sort of thing via a separate app on their phone or tablet, or perhaps through a Web browser that pushes the information to the TV set-top box.

Stylistic Considerations

In addition to structural issues like overscan and focus management, there are some stylistic issues that you will need to take into account when designing your ten-foot UI.

Fonts

With phones and tablets, if the user has some difficulty reading a bit of text, they can usually fix the problem just by moving their hand a bit, to bring the screen closer.

That becomes less likely of a solution as you get into larger screens. People get annoyed if they have to get up off of their sofa to squint and try to read text on a television. In a presentation setting, people may be unable to move into a better viewing position.

To combat this:

- Err on the side of larger fonts, with a medium weight (i.e., not too light or too heavy of strokes that make up the letters)
- Aim to use simpler fonts, particularly sans-serif fonts, as those tend to be more readable at a distance

- Where practical, give the user control over font size within your application, or allow some other sort of "zoom" mechanism, to help see details that they might otherwise be unable to see
- Light text on a dark background tends to be easier to read on televisions, so consider using a theme that supports this (e.g., `Theme.Holo` as opposed to `Theme.Holo.Light`)
- Use fewer words
- Use more line spacing (e.g., via `android:lineSpacingMultiplier` on a `TextView`), so descenders from one line are clearly distinguished from the tops of characters on the next line

Padding and Margins

In addition to adding more line spacing, consider adding more padding and margins to your ten-foot UIs.

Bear in mind that screen density calculations start to go astray as the user moves further away from the screen. We are used to making those calculations based on actual pixels on phones and tablets (a.k.a., "two-foot UI"). The *apparent* size of a television may be no bigger than that of a tablet, once the user's distance from the screen is taken into account. However, screen density has no good way to take that distance into account, other than by effectively hard-coding the density (e.g., Fire TV considering everything to be `-xhdpi`). Hence, particularly for padding and margins, you may need to "finesse" your values a bit on televisions and the like.

In cases where Android is directly talking to the television (e.g., HDMI/MHL from a phone or tablet, Fire TV, Android TV, Android HDMI sticks), you can use `-notouch` qualifiers on resource sets to provide different values for dimension resources.

Colors

Usually, with the ten-foot UI, we treat televisions, monitors, and projectors equally. They do differ in one key area: color management.

Televisions, for historical reasons, tend to have different color responses than do monitors or projectors. As Google puts it:

> TV screens have higher contrast and saturation levels than computer monitors

Even to the extent that those settings could be adjusted, if the television will be used *as a television*, the factory settings may be the proper ones. Beyond that, few television owners think about changing such things.

As a result, you need to be careful with your color choices:

- Pure white (`#FFFFFF`) can cause problems, such as "ghosting", so use a very light gray (e.g., `#F1F1F1`, `#EBEBEB`) instead. Pure black (`#000000`) is not a problem.
- Aim for more muted colors, particularly in the blue/green/violet end of the color spectrum, as opposed to bright red or orange. Warm colors tend to bleed more than cool colors.

Aspect Ratio

Bear in mind that different TVs (or other displays) may have different aspect ratios. While many will be 16:9, also consider 4:3 and 21:9 (also known as 2.35:1).

The Leanback UI

In 2014, Google added the `leanback-v17` library to the Android Support package. This contains code to help you create TV-focused user interfaces. While the intention is for this library to help you create UIs for Android TV, there is nothing strictly tied to the Android TV platform in `leanback-v17`. Your user interfaces can work just fine on other TV environments (e.g., Amazon Fire TV). And they still support touchscreen events, and so they can be used on phones and tablets as well, though perhaps not optimally.

Where to Get Leanback

Android Studio users can add a dependency on the `leanback-v17` artifact from the Android Support Repository:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.android.support:leanback-v17:21.0.3'
}

android {
```

```
    compileSdkVersion 21
    buildToolsVersion '26.0.2'

    defaultConfig {
        applicationId "com.commonsware.android.video.browse"
    }
}
```

(from Leanback/VideoBrowse/app/build.gradle)

This particular bit of Gradle configuration comes from the `Leanback/VideoBrowse` sample application, which will be the focus of this "leanback" UI section. This project depends not only upon `leanback-v17`, but also upon the Picasso image loading library, profiled in the chapter on Internet access.

If you read through the chapter on `MediaStore`, this sample app will seem familiar. In the `MediaStore` chapter, we created a sample app that would present a list of videos available on the device in a `ListView`, using Picasso for handling the video thumbnails. The `VideoBrowse` "leanback" sample app is the same app, adjusted to use a "leanback" UI instead of a `ListView`.

BrowseFragment

The primary UI element that we get from `leanback-v17` is `BrowseFragment`. `BrowseFragment` is a fragment designed to allow browsing of a roster of content through a two-dimensional scrolling interface. There is a list of "headers" (e.g., categories of videos), and within each header is a horizontal scrolling list of items within that header.

This sort of UI pattern is fairly commonplace in TV-centric apps, as it works well with TV-style remotes:

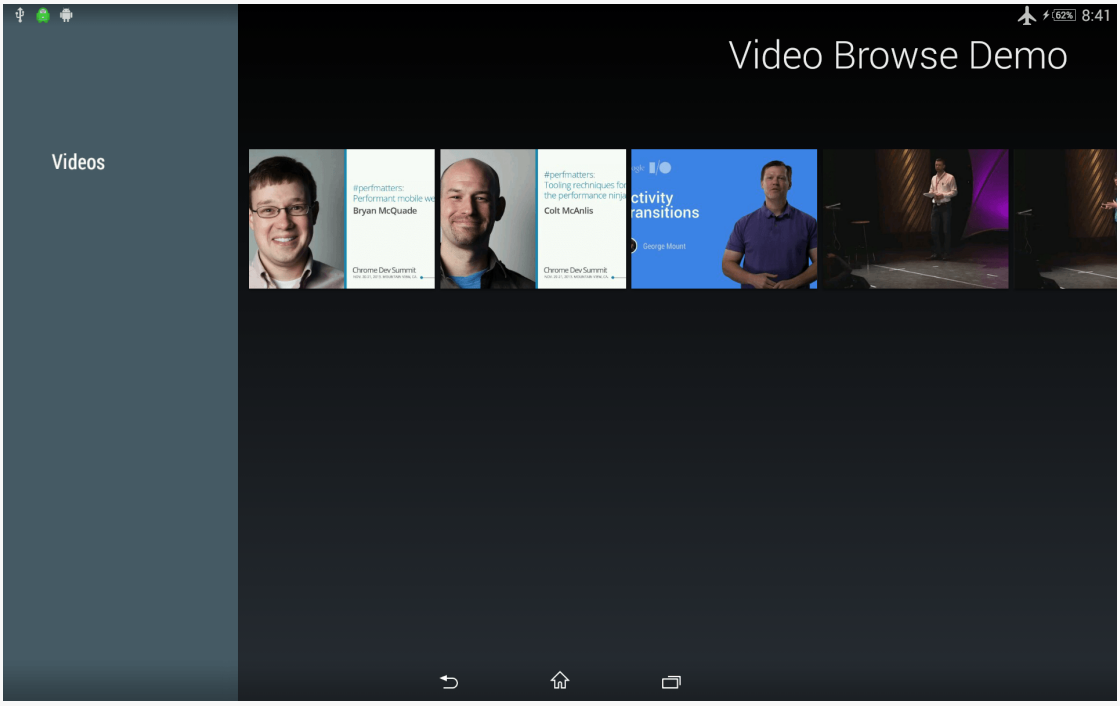


Figure 845: VideoBrowse Sample App, As Initially Launched

The `VideoBrowse` sample application consists of one activity, hosting a `BrowseFragment`, that will display the roster of available videos on the device. Clicking on an individual video will bring up the device's default video player app, just as the `VideoList` sample did in the chapter on the `MediaStore`.

Theme and Activity

There is very little specifically required of an activity that hosts a `BrowseFragment`, particularly on the Java side. So long as the activity gets the `BrowseFragment` onto the screen, the key work is done.

In the case of our `MainActivity`, it uses a `res/layout/main.xml` file, pointing to our `VideosFragment`, which is a subclass of `BrowseFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/videos"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.video.browse.VideosFragment"
/>
```

(from Leanback/VideoBrowse/app/src/main/res/layout/main.xml)

The Java code simply loads up the layout containing that static fragment, plus has an `onVideoSelected()` method that will be called if the user clicks on a video:

```
package com.commonsware.android.video.browse;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import java.io.File;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void onVideoSelected(String uri, String mimeType) {
    Uri video=Uri.fromFile(new File(uri));
    Intent i=new Intent(Intent.ACTION_VIEW);

    i.setDataAndType(video, mimeType);
    startActivity(i);
  }
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/MainActivity.java)

However, there are two other requirements of the activity, in terms of what goes in the manifest:

- The activity needs to use a theme that is, or inherits from, `Theme.Leanback`
- To show up as the "launcher activity" for an Android TV device, the activity needs to have an `<action>` of `MAIN` and a `<category>` of `LEANBACK_LAUNCHER`:

```
  <application
    android:allowBackup="false"
    android:hardwareAccelerated="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="MainActivity"
      android:label="@string/app_name"
      android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSize"
      android:screenOrientation="sensorLandscape"
      android:theme="@style/Theme.Leanback">
```

```
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>

      <category android:name="android.intent.category.LAUNCHER"/>
      <category android:name="android.intent.category.LEANBACK_LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

(from Leanback/VideoBrowse/app/src/main/AndroidManifest.xml)

In our case, we match on either the `LAUNCHER` or the `LEANBACK_LAUNCHER` category, as this particular activity can work on either form factor family (touchscreens or TVs). However, other apps might have separate "launcher activity" implementations for phones/tablets versus televisions, and so having a separate `LEANBACK_LAUNCHER` category allows us to indicate which activities serve which role.

This activity also sets its `screenOrientation` to `sensorLandscape`, indicating that it will always present itself in landscape mode, no matter how the device is held. It also uses a `configChanges` attribute to opt out of configuration changes due to orientation changes, as the UI is not changing in those cases.

Loading the Videos

`VideosFragment` is responsible for showing the roster of available videos on the device, using a `BrowseFragment` two-dimensional structure. This means, though, that `VideosFragment` needs to be able to find out what videos are available. As with the `VideoList` sample in the `MediaStore` chapter, `VideosFragment` will query the `MediaStore` `ContentProvider` to find out about the videos, by means of a `CursorLoader`.

In `onViewCreated()`, `VideosFragment` calls `initLoader()` to start loading the videos, in addition to indicating that the fragment itself will serve as the controller handling clicks on individual videos, via the `setOnItemViewClickedListener()` interface:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  getLoaderManager().initLoader(0, null, this);
  setOnItemViewClickedListener(this);
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

For those calls to work, `VideosFragment` needs to implement the `LoaderManager.LoaderCallbacks<Cursor>` interface (for `initLoader()`) and the `OnItemViewClickedListener` (for `setOnItemViewClickedListener()`).

The `initLoader()` call triggers a call to our `onCreateLoader()` method, which queries the `MediaStore` roster of videos for all videos, ordered by title:

```
@Override
public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
  return(new CursorLoader(
                          getActivity(),
                          MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
                          null, null, null,
                          MediaStore.Video.Media.TITLE));
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

That, in turn, will eventually trigger a call to `onLoadFinished()`:

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
  mapCursorToModels(c);

  setHeadersState(BrowseFragment.HEADERS_ENABLED);
  setTitle(getString(R.string.app_name));

  ArrayObjectAdapter rows=new ArrayObjectAdapter(new ListRowPresenter());
  ArrayObjectAdapter listRowAdapter=
      new ArrayObjectAdapter(new VideoPresenter(getActivity()));

  for (Video v : videos) {
    listRowAdapter.add(v);
  }

  HeaderItem header=new HeaderItem(0, "Videos", null);

  rows.add(new ListRow(header, listRowAdapter));
  setAdapter(rows);
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

We will get to much of the code in `onLoadFinished()` a bit later in this chapter. However, the first thing that `onLoadFinished()` does is call a `mapCursorToModels()` method. This method will be responsible for taking the data from the `Cursor` we get

back from `MediaStore` and using it to populate some model objects that will drive what the `BrowseFragment` displays to the user. `BrowseFragment`'s API is not especially well-suited for working with a `Cursor` directly; it is simpler to have a separate collection of model objects representing the results of the database query.

In our case, the model object will be a `Video`:

```
package com.commonsware.android.video.browse;

class Video {
  int id;
  String uri;
  String mimeType;
  String title;

  Video(int id, String uri, String mimeType, String title) {
    this.id=id;
    this.uri=uri;
    this.mimeType=mimeType;
    this.title=title;
  }

  @Override
  public String toString() {
    return(title);
  }
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/Video.java)

There are four pieces of data we need to track for the video:

- Its unique `id`, so we can get a thumbnail of the video later on
- Its `Uri` (here, held in a string representation), to be used to play back the video
- Its MIME type, also to be used to play back the video
- Its title, which will be used along with its thumbnail when rendering the video as part of the `BrowseFragment` roster of content

`VideosFragment` holds onto a collection of these `Video` objects in a data member named `videos`:

```
  private ArrayList<Video> videos=new ArrayList<Video>();
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

`mapCursorToModels()` iterates over the `Cursor` rows and creates a `Video` object for each row, adding the `Video` to the `videos`, and closing the `Cursor` when done:

```
private void mapCursorToModels(Cursor c) {
  videos.clear();

  int idColumn=c.getColumnIndex(MediaStore.Video.Media._ID);
  int uriColumn=c.getColumnIndex(MediaStore.Video.Media.DATA);
  int mimeTypeColumn=
      c.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);
  int titleColumn=
      c.getColumnIndex(MediaStore.Video.Media.TITLE);

  for (c.moveToFirst(); !c.isAfterLast(); c.moveToNext()) {
    videos.add(new Video(c.getInt(idColumn),
                         c.getString(uriColumn),
                         c.getString(mimeTypeColumn),
                         c.getString(titleColumn)));
  }

  c.close();
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

Headers and Contents

So, let's look at the full `onLoadFinished()` method, called when we have our `Cursor` of videos:

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
  mapCursorToModels(c);

  setHeadersState(BrowseFragment.HEADERS_ENABLED);
  setTitle(getString(R.string.app_name));

  ArrayObjectAdapter rows=new ArrayObjectAdapter(new ListRowPresenter());
  ArrayObjectAdapter listRowAdapter=
      new ArrayObjectAdapter(new VideoPresenter(getActivity()));

  for (Video v : videos) {
    listRowAdapter.add(v);
  }

  HeaderItem header=new HeaderItem(0, "Videos", null);
```

```
    rows.add(new ListRow(header, listRowAdapter));
    setAdapter(rows);
  }
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

As mentioned, the first thing that we do is map the `Cursor` contents to `Video` objects.

We then make two general changes to the look of the `BrowseFragment`:

- We enable the headers. In truth, that would not make sense for this particular application, as we only have one header. However, we are enabling the headers to show what that looks like, as many uses of `BrowseFragment` will need the full two-dimensional browsing experience.
- We set the title to be the `app_name` string resource. This title goes in the upper-right corner and is used to remind the user of what app they are in, much like the title in an action bar would on a phone or tablet.

We then build up the two-dimensional data model and rendering rules for the browsing experience. This involves creating instances of an `ObjectAdapter` base class, supplied by `leanback-v17`. `ObjectAdapter` fills a role reminiscent of `Adapter` with `AdapterView`, insofar as it organizes model data and helps with the rendering. However, whereas `Adapter` does that all itself, `ObjectAdapter` splits the roles out: it handles the model data and delegates to `Presenter` implementations for rendering individual items from the model data.

In the two-dimensional browsing model, we need an `ObjectAdapter` that represents our rows, where each row has a header and a nested `ObjectAdapter` for the items to appear in that row.

Just as `ArrayAdapter` is the easiest `Adapter` class to use, `ArrayObjectAdapter` is the easiest `ObjectAdapter` to use. `ArrayObjectAdapter` adapts arrays of objects, where in this case, "array" really means `ArrayList`.

Unlike `ArrayAdapter`, where we primarily build up our array and hand it to the adapter, `ArrayObjectAdapter` has us populate the "array" via methods like `add()` on the `ArrayObjectAdapter`.

So, after calling `setHeadersState()` and `setTitle()` as described above, we:

- Create an `ArrayObjectAdapter`, named `rows`, that uses the `ListRowPresenter` supplied by `leanback-v17` to render the row
- Create another `ArrayObjectAdapter`, named `listRowAdapter`, that uses a custom `VideoPresenter` that we will examine later in this chapter
- Iterate over the `Video` roster and add each to the `listRowAdapter`
- Create an instance of a `HeaderItem`, supplied by `leanback-v17`, that represents the header entry itself, with a title for that header
- Create an instance of a `ListRow`, supplied by `leanback-v17`, that wraps around the `HeaderItem` and the `listRowAdapter` for the items to show in that row
- Put the `ListRow` in the `rows` `ArrayObjectAdapter`, and pass `rows` to `setAdapter()` to tell the `BrowseFragment` what to display

A more complex app might have several `ListRow` objects in `rows`, one per header. For example, you might group videos by some sort of categorization scheme, where each `HeaderItem` names the category and the `ListRow` also contains the videos specific to that category.

Of the classes cited here, all are stock implementations from `leanback-v17`, with the exception of `VideoPresenter`, which is responsible for rendering a `Video` as an item in the horizontal list of videos.

Presenting the Presenters

A `Presenter`, in the `leanback-v17` system, is an object responsible for converting some model object (e.g., a `Video`) into a visual representation that will be used for an `ObjectAdapter`.

The `Presenter` abstract class enforces a "view holder" approach. A view holder is simply a data structure holding onto a basket of widgets. The idea is that the view holder represents all the widgets for a particular instance of the `Presenter`. So, we now have two levels of indirection over the `Adapter` approach used by `ListView` and kin: not only does `ObjectAdapter` not do the rendering, but `Presenter` alone does not do the rendering, but instead involves a view holder.

As a result, a `Presenter` implementation will tend to be artificially complex.

First, let's look at the view holder, implemented as a static class inside `VideoPresenter`, named `Holder`, that extends the stock `Presenter.ViewHolder` class:

```
static class Holder extends Presenter.ViewHolder {
  private final ImageCardView cardView;
  private int targetWidth, targetHeight;

  public Holder(View view) {
    super(view);

    cardView=(ImageCardView)view;

    Resources res=view.getContext().getResources();

    targetWidth=(int)res.getDimension(R.dimen.card_width);
    targetHeight=(int)res.getDimension(R.dimen.card_height);
  }

  protected void updateCardViewImage(Uri uri) {
    Picasso.with(cardView.getContext())
      .load(uri)
      .resize(targetWidth, targetHeight)
      .centerCrop()
      .onlyScaleDown()
      .placeholder(R.drawable.ic_media_video_poster)
      .into(new Target() {
        @Override
        public void onBitmapLoaded(Bitmap bitmap,
                                   Picasso.LoadedFrom from) {
          Drawable bmpDrawable=
            new BitmapDrawable(
              cardView.getContext().getResources(),
              bitmap);

          cardView.setMainImage(bmpDrawable);
        }

        @Override
        public void onBitmapFailed(Drawable errorDrawable) {
          cardView.setMainImage(errorDrawable);
        }

        @Override
        public void onPrepareLoad(Drawable placeHolderDrawable) {
          cardView.setMainImage(placeHolderDrawable);
        }
      });
  }
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideoPresenter.java)

The `Presenter` will set up the UI, in the form of an `ImageCardView` – another stock class provided by `leanback-v17` that is an `ImageView` with an associated caption. As we will want to pour video thumbnails into the `ImageView`, this sample app uses Picasso, the same way the `VideoList` sample does.

We also determine how big the thumbnail should be, based on a pair of dimension resources, `card_width` and `card_height`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="card_width">400dp</dimen>
  <dimen name="card_height">300dp</dimen>
</resources>
```

(from Leanback/VideoBrowse/app/src/main/res/values/dimens.xml)

The `updateCardViewImage()` will be called when we are ready to use this `ViewHolder` to present the contents of some particular `Video`. We receive the `Uri` to the video and set up Picasso to:

- Load the image from that `Uri` (`load()`)
- Resize it to the target dimensions (`resize()`), but only if the image is larger (`onlyScaleDown()`), and then crop to get the image centered in the desired size (`centerCrop()`)
- Use the supplied `ic_media_video_poster` as a placeholder

Then, since Picasso has no knowledge of how to work with an `ImageCardView` from `leanback-v17`, we have to use a different version of `into()`, one that takes a `Target` as parameter. Here, we use an anonymous inner class implementation of a `Target`. The key method is `onBitmapLoaded()`, where we wrap the `Bitmap` in a `BitmapDrawable` and call `setMainImage` on the `ImageCardView` to populate it. Similarly, there are `onBitmapFailed()` and `onPrepareLoad()` methods for handling errors and the placeholder, respectively.

Now, given the `Holder`, we can set up the rest of `VideoPresenter`, starting with its constructor:

```
  VideoPresenter(Context ctxt) {
    super();

    this.ctxt=ctxt;
  }
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideoPresenter.java)

Here, mostly, we are holding onto a supplied `Context` for eventually creating our `ImageCardView`.

At the point in time that the `VideoPresenter` needs to create a view holder to use for rendering an item, `onCreateViewHolder()` will be called:

```
@Override
public ViewHolder onCreateViewHolder(ViewGroup parent) {
  ImageCardView cardView=new ImageCardView(ctxt);

  cardView.setFocusable(true);
  cardView.setFocusableInTouchMode(true);

  return(new Holder(cardView));
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideoPresenter.java)

Here, we set up an `ImageCardView`, marking it as focusable both for the D-pad and for touchscreens, and wrap that in our custom `Holder`.

At the point in time when we are ready to show a `Video` using the widgets managed by the `Holder`, `onBindViewHolder()` is called:

```
@Override
public void onBindViewHolder(Presenter.ViewHolder viewHolder,
                             Object item) {
  Video video=(Video)item;
  Holder h=(Holder)viewHolder;
  Resources res=ctxt.getResources();

  h.cardView.setTitleText(video.toString());
  h.cardView.setMainImageDimensions((int)res.getDimension(R.dimen.card_width),
                                    (int)res.getDimension(R.dimen.card_height));

  Uri thumbnailUri=
      ContentUris.withAppendedId(MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
          video.id);

  h.updateCardViewImage(thumbnailUri);
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideoPresenter.java)

We are passed in a generic `Object` that is the model data from our `ObjectAdapter` — in this case, it will be a `Video`, as we are using `VideoPresenter` with an `ArrayObjectAdapter` that holds the `Video` instances. We update the `ImageCardView` caption based on the title of the `Video`, then set the size of the `ImageView` based upon the same dimension resources as we used with UIL's `ImageSize`. We construct

a `Uri` pointing to the video as known to `MediaStore` (given the video's id), and pass the `String` representation of that into the `Holder`, which will handle the UIL work.

There is one other abstract method that we need to override to satisfy `Presenter`: `onUnbindViewHolder()`:

```
@Override
public void onUnbindViewHolder(Presenter.ViewHolder viewHolder) {
  ((Holder)viewHolder).cardView.setMainImage(null);
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideoPresenter.java)

`onUnbindViewHolder()` can often be skipped. However, if you have memory-intensive stuff in the view holder — like the bitmap in an `ImageView` — this is a fine time to take steps to release that memory. In our case, we `null` out the image in the `ImageCardView`. Ideally, we would somehow work with UIL to arrange to recycle this `Bitmap` object, since all of our `Bitmap` objects should be the same size.

Handling Clicks

`BrowseFragment` automatically handles a lot of input events, such as:

- arrow key events in the list of headers, to move a selection bar up and down the list
- click events on a header, to allow navigation into the row of items for that header
- arrow key events on an item, to navigate to the next or previous item
- BACK button events on an item, to return to the list of headers

It also captures the click event on an item and routes that to the `onItemClicked()` method of the `BrowseFragment`, which we override in `VideosFragment`:

```
@Override
public void onItemClicked(Presenter.ViewHolder viewHolder,
                          Object o,
                          RowPresenter.ViewHolder rowViewHolder,
                          Row row) {
  Video video=(Video)o;
  ((MainActivity)getActivity()).onVideoSelected(video.uri,
                                                video.mimeType);
}
```

(from Leanback/VideoBrowse/app/src/main/java/com/commonsware/android/video/browse/VideosFragment.java)

The `Object` passed as the second parameter to `onItemClicked()` is the `Object` in our `ObjectAdapter` for the clicked-upon item. In our case, the `ObjectAdapter` is our `ArrayObjectAdapter` wrapped around our `Video` objects, and so the `Object` passed into `onItemClicked()` is a `Video`. Given that, we can call out to the hosting activity and its `onVideoSelected()` to go play back the selected video.

The Results

Launching the app shows our list of headers (with just the one header), thumbnails of videos in that header, and the "Video Browse Demo" title:

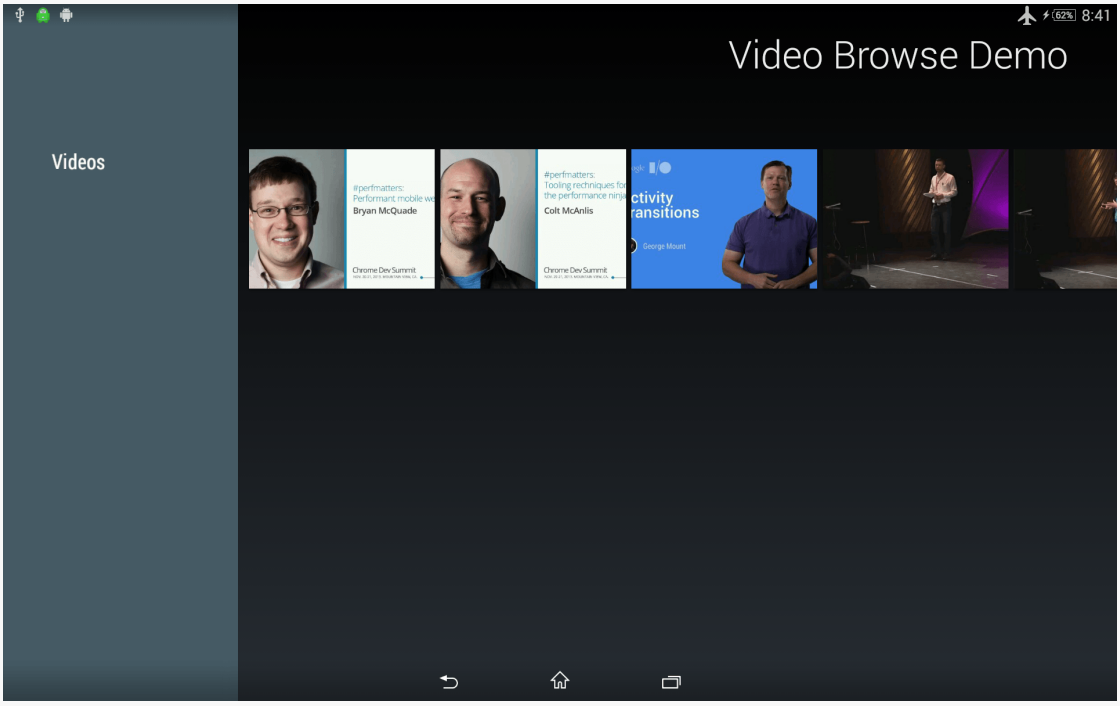


Figure 846: VideoBrowse Sample App, As Initially Launched

Selecting a video slides the headers out of the way and shows the full card for the video, including the video's title:

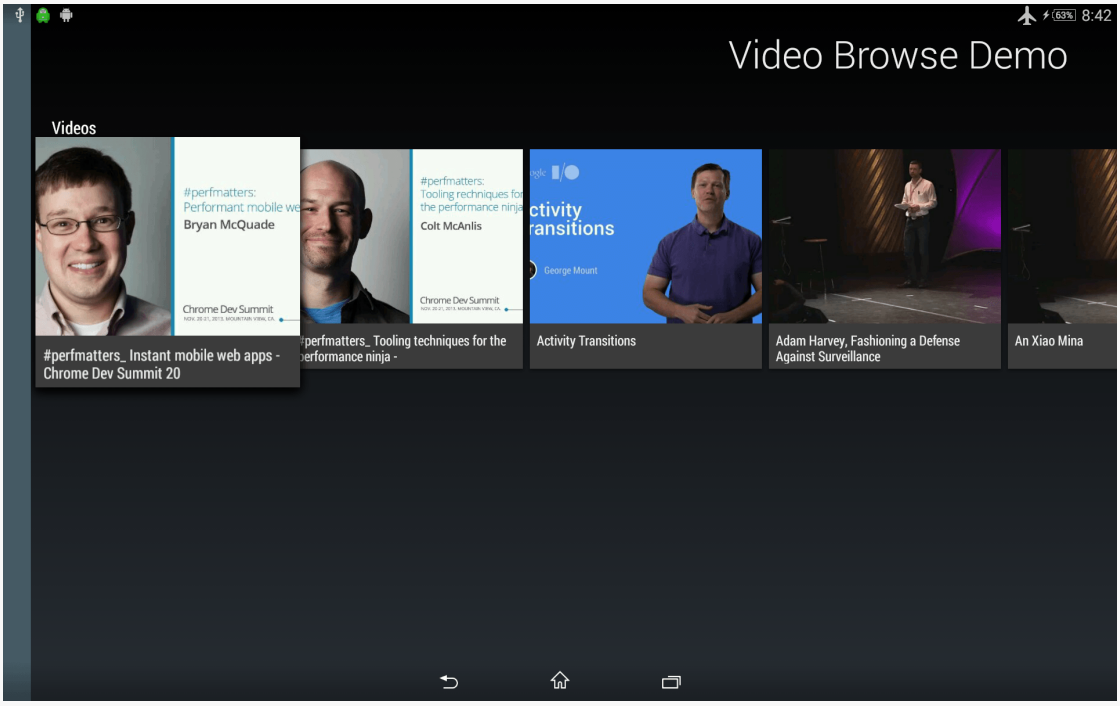


Figure 847: VideoBrowse Sample App, With First Video Selected

Clicking on the selected video brings up the default video player for the device.

Note that this UI is not tied strictly to TV-style displays. For example, the screenshots shown in this section came from an Android tablet, as you can tell by the status bar and navigation bar. The `BrowseFragment` UI is not completely out of place on a tablet, and it works with touch events as well as the key events that would be emitted by a TV-style remote control. On a phone, the `BrowseFragment` UI gets a bit cramped, particularly in portrait, though it still works.

Testing Your Theories

Ideally, you test your ten-foot UI in a ten-foot experience, using something connected to a television.

This does not have to be expensive:

- If you have a phone or tablet that can connect to a TV via HDMI, MHL, or Miracast, at most you might need a cable or a Miracast adapter

- Android "HDMI sticks" or other Android set-top boxes can be found on eBay, Alibaba, or elsewhere
- Chromecast and the Fire TV Stick are available in some markets fairly inexpensively

Any of those will cost $10-75. At the $100 price point, you can start looking at the Fire TV or Nexus Player as well.

If you need to develop using more traditional hardware (phone or a tablet) or an emulator, the big thing will be to make sure that you are estimating the screen size properly. For example, a 10" tablet, held fully at arm's length, will have the same visual angle as a modest television (~26") at a comfortable seating distance. While this will not help with color saturation, using remote controls, or other aspects of the ten-foot UI, you can at least get a sense of whether your text and UI controls will be large enough to be usable.

Putting the TVs All Together: Decktastic

This book profiles many ways of getting content to a TV:

- By means of `Presentation` and related classes, for touchscreen-enabled devices that also happen to presently have a connection to an external display
- By means of `RemotePlaybackClient`, for use with devices like the Chromecast
- By means of directly displaying output on a TV, for devices where the TV is the primary display (e.g., Android TV devices, Amazon's Fire TV and Fire TV Stick)

It is entirely possible to create one app that can support all of these modes from one code base, though you are constrained by the most limited option. In this case, `RemotePlaybackClient` is the most limited option, as its API is designed to tell some external device to play some media, whereas the other options can support comparatively arbitrary user interfaces rendered through normal Android widgets.

In this chapter, we will review the `Presentation/Decktastic` sample application. This app is designed to give the user a roster of slide-based presentations to choose from, then deliver one of those presentations. The presentation will appear on the external display (e.g., TV or projector), while the presenter will be able to control the presentation either from a touchscreen-equipped Android device or a remote control.

Prerequisites

You should read the following chapters before this one:

- Supporting External Displays
- Media Routes
- The "10 Foot UI"

Reading up on specific hardware, like the Amazon Fire TV, is a good idea but not as critical.

Introducing Decktastic

Before we get into discussing the implementation of Decktastic, we should first review what the app looks like and how it functions.

Launcher UI

If you were to set up Decktastic on some test device and run it, the first thing that you would see is a media browsing UI built from the `leanback-v17` support library, showing you a roster of the available presentations to choose from:

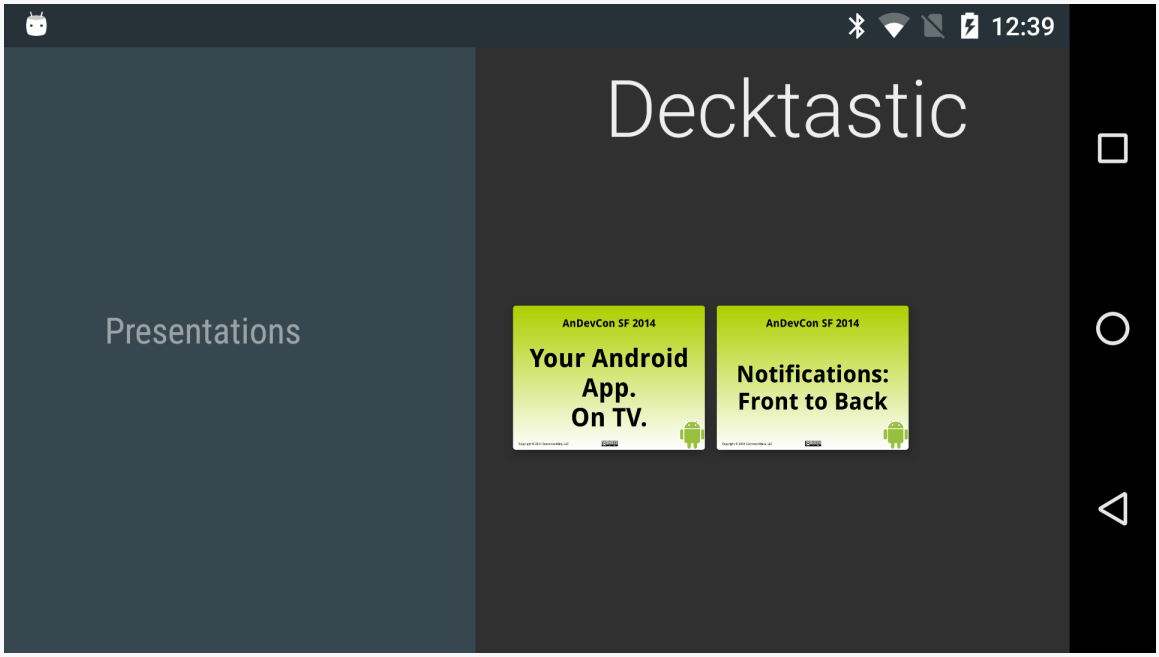


Figure 848: Decktastic Media Browser

This UI works fine on TVs and on tablets. On phones... it gets a bit cramped.

Tapping on a presentation selects it:

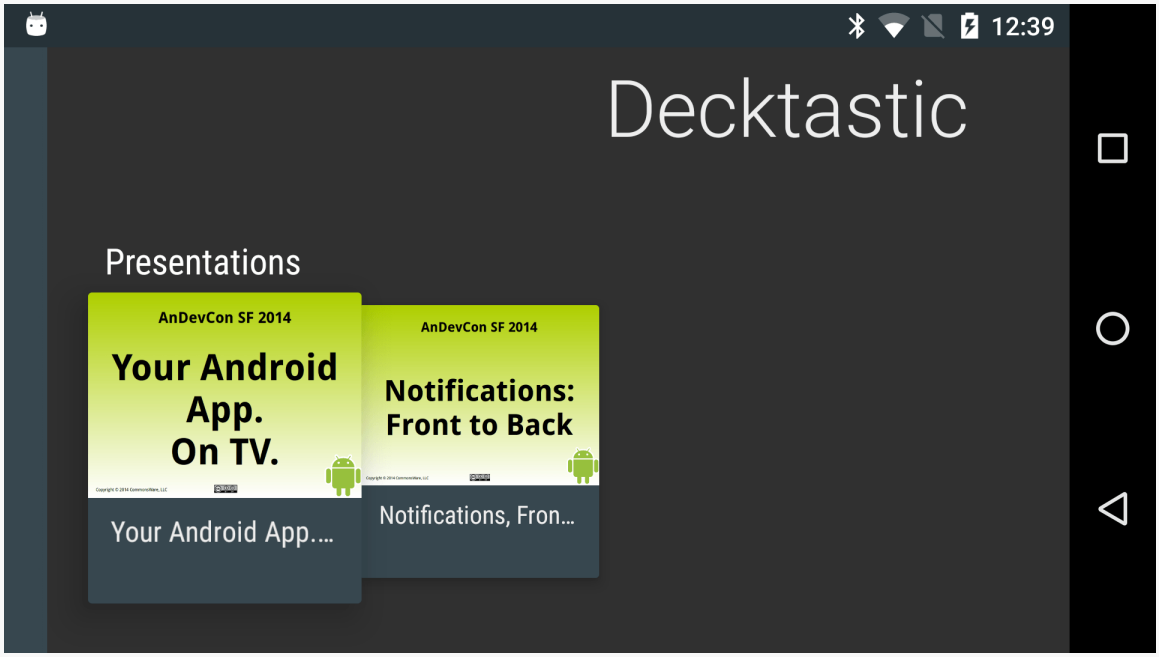


Figure 849: Decktastic Media Browser, With Selected Presentation

Presentation UI

Tapping on the presentation again opens it up into a `ViewPager`-based UI for the presenter:

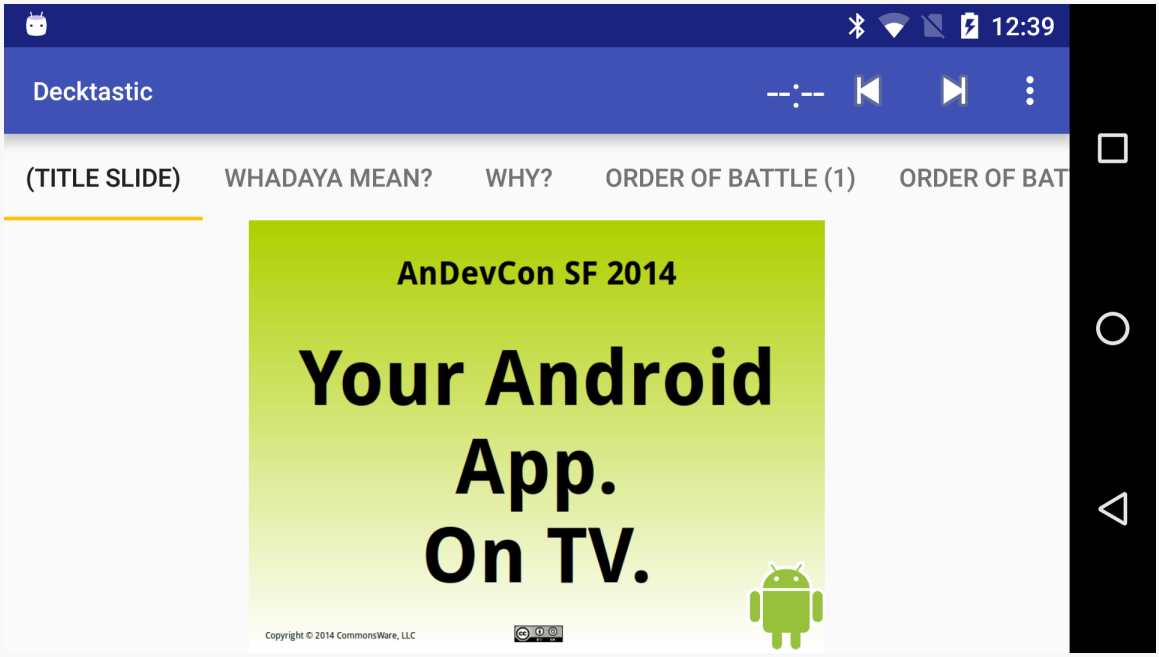


Figure 850: Decktastic Main UI, Showing Presentation and Open Overflow

However, who sees what depends a bit upon the available hardware:

- If you are running this standalone on a phone or tablet, you will see the `ViewPager`-based UI
- If you are running this on a phone or tablet with a connection to an external display, *you* will see the `ViewPager`-based UI, but the *audience* (those looking at the external display) will, by default, just see the slides
- If you are running this on a phone or tablet with a connection to a Chromecast or similar remote playback device, we get the same results as with the external display (you see the full UI, the audience sees the slides)
- If you are running this on an Android TV, Fire TV, or similar device, both you and the audience only see the slides

To move through the slides, you can:

- Swipe the `ViewPager`
- Use the `ViewPager` tabs
- Use right or down keys to move forward, or left or up keys to move backward, whether on a QWERTY keyboard (e.g., Bluetooth) or via the D-pad on some form of remote (for TV-centric scenarios, like Android TV or Fire TV)

Implementing Decktastic

Decktastic is a pair of activities that "stand upon the shoulders of giants", in the form of using seven third-party libraries that provide a lot of the utility code.

The Gradle Dependencies

The project's `build.gradle` file specifies a fair number of dependencies:

```
dependencies {
    implementation 'com.android.support:leanback-v17:24.1.1'
    implementation 'com.android.support:mediarouter-v7:24.1.1'
    implementation 'com.android.support:design:24.1.1'
    implementation 'com.google.code.gson:gson:2.7'
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.commonsware.cwac:presentation:0.4.5'
}
```

(from Presentation/Decktastic/app/build.gradle)

The project uses:

- `cwac-presentation` for `PresentationHelper`
- `design`, for its `TabLayout`; this library also pulls in `appcompat-v7` (for `AppCompatActivity` and kin) and `support-v4` (for `ViewPager` and kin)
- Google's Gson
- greenrobot's EventBus
- Google's `leanback-v17` library, for "ten-foot UI" elements used in our launcher activity
- Square's Picasso, for asynchronously loading images

The Presentation Format

A Decktastic presentation consists of a JSON file and a series of image files. The image files are the slides, perhaps exported from a traditional presentation package like LibreOffice Impress. The JSON file spells out what the image files are and their order of appearance.

For example, here is a JSON file from one of the presentations:

```
{
    "title": "Notifications, Front to Back",
    "duration": 70,
    "baseURL": "http://misc.commonsware.com/andevcon2014/preso2/",
    "slides": [
        {
            "title": "(title slide)",
            "image": "img0.png"
        },
        {
            "title": "Order of Battle",
            "image": "img1.png"
        }
    ]
}
```

The `title` is used on the initial "leanback" activity as part of displaying the available presentations.

The `duration` is how long the presentation should run, in minutes. This will be used for a countdown timer to help the presenter know how much time remains in the presentation.

The `baseURL` is a URL to a directory on a Web server somewhere that contains the same slide images as are available locally. This is needed to support `RemotePlaybackClient`, as Chromecast and similar devices need to be able to download their content over the network. We do not have an easy way to deliver that content from the phone or tablet that runs Decktastic and so we need a network-hosted copy of the slides as well. If you were willing to dispense with Chromecast support, you would not need this `baseURL` property.

The `slides` array contains JSON objects, each of which provides the title for a slide and the image associated with that slide. The title will be used for the `ViewPager` tabs, so the presenter knows the upcoming slides and can rapidly switch to a specific slide. The images, of course, are what the presenter and the audience see. Of particular importance is the first slide in the array, as this will be used as the "title slide", shown on the initial "leanback" activity.

The JSON and slides are stored as assets. One full presentation ("Your Android App. On TV.") is stored in `assets/preso1/`, while a stub presentation ("Notifications, Front to Back") is stored in `assets/preso2/`.

The Model Classes

Given that JSON, we need a model class that will represent it, caching the parsed JSON so that we can use that information to render the presentation. We also need a model class that represents the collection of parsed presentations, so that we have the information necessary to render the "leanback" activity that allows the user to find the presentation to display.

There are two model classes in the book that handle this: `PresoContents` and `PresoRoster`.

PresoContents

`PresoContents` represents the parsed JSON, along with a few other bits of information about the presentation:

```
package com.commonsware.android.preso.decktastic;

import java.util.List;

public class PresoContents {
  String title;
  List<Slide> slides;
  int duration;
  String baseURL;
  String baseDir;
  int id=-1;

  static class Slide {
    String image;
    String title;
  }

  @Override
  public String toString() {
    return(title);
  }

  String getSlideImage(int position) {
    return(baseDir+slides.get(position).image);
  }

  String getSlideTitle(int position) {
    return(slides.get(position).title);
  }
```

```
  String getSlideURL(int position) {
    return(baseURL+slides.get(position).image);
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoContents.java)

The `title`, `slides`, `duration`, and `baseURL` fields come straight from the JSON. The `baseDir` field represents the directory in which the presentation was loaded; all images will be assumed to be relative to this directory. Finally, each presentation is given an `id`, so we can distinguish one presentation from another in our collection of presentations.

`PresoContents` also has getter methods to retrieve the local image file (`getSlideImage()`), title (`getSlideTitle()`), and remote image URL (`getSlideURL()`) for a slide given its `position` in the array of slides.

PresoRoster

`PresoRoster` is a singleton collection of the available presentations. It also contains the model logic for loading the collection of presentations and parsing the JSON to create an individual `PresoContents` object for a single presentation:

```
package com.commonsware.android.preso.decktastic;

import android.content.Context;
import android.content.res.AssetManager;
import android.util.Log;
import com.google.gson.Gson;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

class PresoRoster {
  private static final PresoRoster INSTANCE=new PresoRoster();
  private static String[] PRESO_ASSET_DIRS={"preso1/", "preso2/"};
  private List<PresoContents> presos=new ArrayList<PresoContents>();

  static PresoRoster getInstance() {
    return(INSTANCE);
  }
```

```
private PresoRoster() {}

int getPresoCount() {
  return(presos.size());
}

PresoContents getPreso(int position) {
  return(presos.get(position));
}

PresoContents getPresoById(int id) {
  return(getPreso(id));
}

void load(Context ctxt) {
  Gson gson=new Gson();
  AssetManager assets=ctxt.getAssets();

  for (String presoDir : PRESO_ASSET_DIRS) {
    PresoContents c=loadPreso(gson, assets, presoDir);

    if (c!=null) {
      c.id=presos.size();
      presos.add(c);
    }
  }
}

private PresoContents loadPreso(Gson gson, AssetManager assets,
                                String presoDir) {
  PresoContents result=null;

  try {
    InputStream is=assets.open(presoDir+"preso.json");
    BufferedReader reader=
        new BufferedReader(new InputStreamReader(is));

    result=gson.fromJson(reader, PresoContents.class);
    result.baseDir=presoDir;
    is.close();
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
  }

  return(result);
```

```
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoRoster.java)

The class includes:

- Getters to retrieve a presentation by index in the array of presentations (`getPreso()`) and to retrieve a presentation by the ID of the presentation (`getPresoById()`)
- A `getPresoCount()` method that indicates how many presentations were found
- A `load()` method that will iterate over known presentation asset directories (defined in `PRESO_ASSET_DIRS`), then attempt to parse a `preso.json` file in the asset directory for a presentation (via a private `loadPreso()` method) using Gson

The result of `load()` is that the `PresoRoster` should be populated with all known presentations in assets. Note, though, that this work is done on the current thread, and therefore `load()` needs to be called on a background thread. Also note that `PresoRoster` makes no attempt at thread synchronization, and so `load()` should be called before anything attempts to use the `PresoRoster` getter methods like `getPresoCount()`.

The Launcher Activity: LeanbackActivity

As noted previously, our launcher activity is one that implement's Google's "leanback" user interface, specifically a `BrowseFragment` for browsing media content. In this case, that content consists of the roster of available presentations.

The `LeanbackActivity` itself is fairly short:

```
package com.commonsware.android.preso.decktastic;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class LeanbackActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager()
          .beginTransaction()
          .add(android.R.id.content, new RosterFragment()).commit();
    }
  }

  public void showPreso(PresoContents preso) {
    startActivity(new Intent(this, MainActivity.class)
                    .putExtra(MainActivity.EXTRA_PRESO_ID,
                              preso.id));
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/LeanbackActivity.java)

All it does is add a `RosterFragment` to the UI managed by the activity, plus add a `showPreso()` method that will be called by that `RosterFragment` when a presentation is selected. `showPreso()`, in turn, will start a separate activity (`MainActivity`), supplying `EXTRA_PRESO_ID` with the ID of the selected presentation, so `MainActivity` knows what presentation to show.

Manifest Entry

To work properly with the `leanback-v17` classes like `BrowseFragment`, `LeanbackActivity` needs to use `Theme.Leanback`, supplied by `leanback-v17`:

```
      <activity
          android:name="com.commonsware.android.preso.decktastic.LeanbackActivity"
          android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSize"
          android:label="@string/app_name"
          android:screenOrientation="sensorLandscape"
          android:theme="@style/Theme.Leanback">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />

          <category android:name="android.intent.category.LAUNCHER" />
          <category android:name="android.intent.category.LEANBACK_LAUNCHER" />
        </intent-filter>
      </activity>
```

(from Presentation/Decktastic/app/src/main/AndroidManifest.xml)

Other noteworthy items in the `<activity>` element in the manifest include:

- Locking the screen orientation to `sensorLandscape`, as we want to stick with a landscape-style orientation, but it could either be “regular” or “reverse” landscape without issue

- Handling orientation-related configuration changes, which are not needed since we are locking the screen orientation to `sensorLandscape`, and therefore the UI does not change on an orientation change
- Having the `LEANBACK_LAUNCHER` category as an option in the `<intent-filter>`, as this will cause this activity to appear on Android TV's home screen launcher (as opposed to the `LAUNCHER` category used by normal Android devices)

RosterFragment

`RosterFragment` is a `BrowseFragment`, designed to provide the two-dimensional navigation of headers and items in a header. In this case, we will have just one header, "Presentations", containing all of the presentations found by `PresoRoster`.

In `onAttach()`, we check to see how many presentations are known about. If there are none, we make two assumptions:

1. That this is the first time we have needed to look for presentations, and
2. That there are presentations to be found

So, we fork a `LoadThread` to go load those presentations:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);

  if (PresoRoster.getInstance().getPresoCount()==0) {
    new LoadThread(host).start();
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/RosterFragment.java)

Of course, those assumptions are a gross simplification. It could be that the user launched our `LeanbackActivity`, pressed BACK, then launched it again for some reason, and therefore the first `LoadThread` did not yet finish before we go and fork a second one. Or, it could be that there are no presentations to be found, in which case we scan unnecessarily. A production-grade version of this app should have a more sophisticated means of ensuring a one- (and only one-) time initialization.

`LoadThread` drops our thread priority to background levels, then tells the `PresoRoster` to load presentations from our app's standard spot on external storage. Then, we raise a `RosterLoadedEvent` on greenrobot's EventBus:

```
private static class LoadThread extends Thread {
  private Context ctxt=null;

  LoadThread(Context ctxt) {
    super();

    this.ctxt=ctxt.getApplicationContext();
  }

  @Override
  public void run() {
    android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);
    PresoRoster.getInstance().load(ctxt);

    EventBus.getDefault().postSticky(new RosterLoadedEvent());
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/RosterFragment.java)

You will notice that we call `postSticky()`, not `post()` on the `EventBus` instance. This says that we not only want to deliver this event to any current registrants, but that the `EventBus` should cache this event and hand it to future registrants.

To respond to the `RosterLoadedEvent`, we register the `RosterFragment` on the bus in `onResume()` and unregister in `onPause()`:

```
@Override
public void onResume() {
  super.onResume();

  EventBus.getDefault().register(this);
}

@Override
public void onPause() {
  EventBus.getDefault().unregister(this);

  super.onPause();
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/RosterFragment.java)

We then have `onRosterLoaded()` set up with the `@Subscribe` annotation to watch for the `RosterLoadedEvent`:

```
@Subscribe(sticky=true, threadMode=ThreadMode.MAIN)
public void onRosterLoaded(RosterLoadedEvent event) {
  setHeadersState(BrowseFragment.HEADERS_ENABLED);
  setTitle(getString(R.string.app_name));

  ArrayObjectAdapter rows=new ArrayObjectAdapter(new ListRowPresenter());
  PresoRoster roster=PresoRoster.getInstance();
  ArrayObjectAdapter listRowAdapter=new ArrayObjectAdapter(new PresoPresenter());

  for (int i=0; i < roster.getPresoCount(); ++i) {
    listRowAdapter.add(roster.getPreso(i));
  }

  HeaderItem header=new HeaderItem(0, "Presentations");
  rows.add(new ListRow(header, listRowAdapter));

  setAdapter(rows);
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/RosterFragment.java)

The `sticky=true` part of the annotation, in conjunction with `postSticky()`, means that if events were sticky-posted in the past, we are delivered those immediately, in addition to future events. This will allow us to handle configuration changes — even though our activity and fragment might be destroyed on a locale change, or if our device is put into some sort of desk dock, we will get the `RosterLoadedEvent` when our fragment is created anew.

The `threadMode=ThreadMode.MAIN` portion of the annotation indicates that we want the event to be received on the main application thread, even though it was raised via a background thread.

In `onRosterLoaded()`, we:

- Indicate that we do want headers (though, in reality, since this app only has one header, you could easily skip the headers)
- Set the title to appear in the upper-right corner
- Create an `ArrayObjectAdapter` for the `rows` that make up the entirety of the `BrowseFragment` contents, using the standard `ListRowPresenter` for our headers and rows
- Create another `ArrayObjectAdapter`, wrapped around a `PresoPresenter`, that will manage the presentations in our one-and-only row
- Pour our `PresoContents` instances into the `ArrayObjectAdapter` for our row
- Attach the “Presentations” title to the row via a standard `ListRow` object
- Tell the `RosterFragment` that the `rows` represents what it should render

This is all covered in greater detail in the chapter on the “ten-foot” UI.

In `onViewCreated()` of `RosterFragment`, we indicate that the `RosterFragment` itself should be the listener for click events on items (in our case, presentations):

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  setOnItemViewClickedListener(this);
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/RosterFragment.java)

This works because `RosterFragment` implements the `OnItemViewClickedListener` interface and therefore implements the `onItemClicked()` method:

```
@Override
public void onItemClicked(Presenter.ViewHolder viewHolder,
                          Object o,
                          RowPresenter.ViewHolder rowViewHolder,
                          Row row) {
  ((LeanbackActivity)getActivity()).showPreso((PresoContents)o);
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/RosterFragment.java)

Here, we ask the hosting `LeanbackActivity` to show the clicked-upon presentation, which causes `LeanbackActivity` to launch a `MainActvity` to do just that.

PresoPresenter

The role of `PresoPresenter` is to render the individual items shown in the `BrowseFragment`. In this case, the items are `PresoContents` model objects; `PresoPresenter` will pour the presentation information into `ImageCardView` widgets. `ImageCardView` is supplied by the `leanback-v17` library and is designed to be used for rendering items in a `BrowseFragment`.

The `Presenter` abstract class — which `PresoPresenter` extends — enforces the view holder pattern. A `Presenter` is really responsible for creating and updating `Presenter.ViewHolder` instances, which in turn are responsible for updating the actual widgets themselves. To that end, the `PresoPresenter.Holder` static class is a subclass of `Presenter.ViewHolder`, one that is responsible for pouring a `PresoContents` into an `ImageCardView`:

```
static class Holder extends Presenter.ViewHolder {
  private ImageCardView cardView;
  private PicassoImageCardViewTarget viewTarget;

  public Holder(View view) {
    super(view);

    cardView=(ImageCardView)view;
    viewTarget=new PicassoImageCardViewTarget(cardView);
  }

  protected void updateCardViewImage(String path) {
    Picasso.with(cardView.getContext())
            .load("file:///android_asset/" + path)
            .resize(convertDpToPixel(cardView.getContext(), CARD_WIDTH),
                convertDpToPixel(cardView.getContext(), CARD_HEIGHT))
            .into(viewTarget);
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

Here, we are going to use Picasso to load the initial slide off of disk and put it in the `ImageCardView`. However, Picasso has no built-in knowledge of `ImageCardView`, the way it has built-in knowledge of `ImageView`. We need to teach Picasso how to populate an `ImageCardView`. Picasso's mechanism for this is to define a `Target` implementation (`PicassoImageCardViewTarget` in this case) that is responsible for taking a loaded bitmap and updating the UI with it:

```
private static class PicassoImageCardViewTarget implements Target {
  private ImageCardView imageCardView;

  public PicassoImageCardViewTarget(ImageCardView imageCardView) {
    this.imageCardView=imageCardView;
  }

  @Override
  public void onBitmapLoaded(Bitmap bmp, Picasso.LoadedFrom lf) {
    Drawable bmpDrawable=
        new BitmapDrawable(imageCardView.getContext().getResources(),
                           bmp);

    imageCardView.setMainImage(bmpDrawable);
  }

  @Override
  public void onBitmapFailed(Drawable d) {
```

```
      imageCardView.setMainImage(d);
    }

    @Override
    public void onPrepareLoad(Drawable d) {
      imageCardView.setMainImage(d);
    }
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

`Target` requires implementations of:

- `onBitmapLoaded()`, where we take the image and put it as the "main image" of the `ImageCardView` by means of `setMainImage()`
- `onBitmapFailed()`, where we are given a failure `Drawable` and need to use it, once again by setting it as the `ImageCardView` main image
- `onPrepareLoad()`, where we are given a "loading" `Drawable` and need to use it, once more by setting it as the `ImageCardView` main image

The `PresoPresenter.Holder` class creates an instance of a `PicassoImageCardViewTarget` and uses that for the `into()` method of the Picasso `RequestBuilder` (created via the `with()` static method on the `Picasso` class).

The other thing interesting about our use of Picasso is in the `resize()` call. Particularly since Picasso does not know about `ImageCardView` and how big the image should be, we need to manually tell Picasso what size to make the image. Here, we hard-code the sizes of the card width and height in density-independent pixels:

```
  private static final int CARD_WIDTH=400;
  private static final int CARD_HEIGHT=300;
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

We then use a static `convertDpToPixel()` method to get the actual number of hardware pixels to use, based upon the current screen density:

```
  static int convertDpToPixel(Context ctxt, int dp) {
    float density=ctxt.getResources().getDisplayMetrics().density;

    return(Math.round((float)dp*density));
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

Back up in `PresoPresenter` itself, the `Presenter` abstract class requires us to override `onCreateViewHolder()`, where we are responsible for creating a `Presenter.ViewHolder`. In the case of `PresoPresenter`, that comes in the form of the aforementioned `PresoPresenter.Holder`:

```
@Override
public ViewHolder onCreateViewHolder(ViewGroup parent) {
  ImageCardView cardView=new ImageCardView(parent.getContext());

  cardView.setFocusable(true);
  cardView.setFocusableInTouchMode(true);

  return(new Holder(cardView));
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

We also have to override `onBindViewHolder()`, where we are given an eligible `Presenter.ViewHolder` and need to populate its widgets from a supplied item:

```
@Override
public void onBindViewHolder(Presenter.ViewHolder viewHolder,
                             Object item) {
  PresoContents preso=(PresoContents)item;
  Holder h=(Holder)viewHolder;

  h.cardView.setTitleText(preso.toString());
  h.cardView.setMainImageDimensions(CARD_WIDTH, CARD_HEIGHT);
  h.updateCardViewImage(preso.getSlideImage(0));
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

Here, the item is a `PresoContents` and the `Presenter.ViewHolder` is a `PresoPresenter.Holder`. We update the `ImageCardView` title and image size based on the presentation, plus tell the `Holder` to update the image itself, calling the `updateCardViewImage()` method that contained our Picasso request.

Note that we are passing the density-independent pixels values (`CARD_WIDTH`, `CARD_HEIGHT`) to `setMainImageDimensions()`. Unfortunately, this method is undocumented, and so what the units of measure should be are not disclosed.

The `Presenter` abstract class also requires implementations of `onUnbindViewHolder()` (called when we should no longer be populating those

widgets) and `onViewAttachedToWindow()` (called when the widgets associated with a `Presenter.ViewHolder` are now "live"):

```
@Override
public void onUnbindViewHolder(Presenter.ViewHolder viewHolder) {
  ((Holder)viewHolder).cardView.setMainImage(null);
}

@Override
public void onViewAttachedToWindow(Presenter.ViewHolder viewHolder) {
  // no-op
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresoPresenter.java)

The Guts: MainActivity

All of the above was just to handle the launcher activity, to allow the user to choose a presentation. `MainActivity` is where we actually show the presentation itself. This is based upon the `Presentation/Slides` sample app. profiled in the chapter on `Presentation`, with a replacement implementation of the tabs, and additional logic to handle `RemotePlaybackClient`-compatible devices (e.g., Chromecast) and TV-centric devices (e.g., Fire TV).

Basic Setup

`onCreate()` of `MainActivity` in responsible for basic setup.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  preso=
    PresoRoster
      .getInstance()
      .getPresoById(getIntent().getIntExtra(EXTRA_PRESO_ID, 0));

  setContentView(R.layout.activity_main);

  pager=(ViewPager)findViewById(R.id.pager);
  helper=new PresentationHelper(this, this);

  selector=
      new MediaRouteSelector.Builder()
          .addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)
```

```
            .build();
    router=MediaRouter.getInstance(this);
    router.addCallback(selector, routeCB,
        MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);

    if (isDirectToTV()) {
      getSupportActionBar().hide();
    }

    setupPager();
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

First, we take the `EXTRA_PRESO_ID` value received via an `Intent` extra and uses that to find the `PresoContents` object representing the presentation to be shown. That `PresoContents` object is then referenced by a data member named `preso`.

Next, we load up the `activity_main` layout resource, containing our `ViewPager` and a `TabLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:keepScreenOn="true"
  android:orientation="vertical">

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.design.widget.TabLayout
      android:id="@+id/tabs"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:visibility="gone"
      app:tabMode="scrollable" />
  </android.support.v4.view.ViewPager>

</LinearLayout>
```

(from Presentation/Decktastic/app/src/main/res/layout/activity_main.xml)

Then, we create a `PresentationHelper`, so that we find out when we should and should not be displaying a `Presentation`. As before, `MainActivity` itself is the `PresentationHelper.Listener` for finding out about these events. We will explore that more later in this chapter.

We then go through some logic for setting up remote playback device support (`MediaRouteSelector.Builder` and kin) and direct-to-TV device support (calling `isDirectToTV()`). Those will be explored later in this chapter, in sections on remote playback device support and direct-to-TV device support.

Finally, we call `setupPager()`, to populate our `ViewPager`.

The ViewPager

The `setupPager()` method is responsible for putting a `SlidesAdapter` into the `ViewPager` and otherwise setting things up to allow the presenter to control what slide is shown and for us to find out what slide the presenter selects:

```
private void setupPager() {
  durationInSeconds=preso.duration * 60;

  if (rc!=null) {
    rc.setOverallDuration(durationInSeconds);
  }

  adapter=new SlidesAdapter(this, preso);
  pager.setAdapter(adapter);

  if (!isDirectToTV()) {
    TabLayout tabs=(TabLayout)findViewById(R.id.tabs);

    tabs.setVisibility(View.VISIBLE);
    tabs.setupWithViewPager(pager);
    tabs.addOnTabSelectedListener(this);
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Some of this — specifically the `SlidesAdapter` logic — is standard `ViewPager` setup work, with `TabLayout` being a popular implementation of tabs for activities that, like this one, extend from `AppCompatActivity`. The `durationInSeconds` stuff at the top is for setting up a `ReverseChronometer`, as will be discussed later in this chapter. The

`isDirectToTV()` call and `if` block will be explained more in the section on direct-to-TV device support later in this chapter.

`SlidesAdapter` is a fragment-free edition of a `PagerAdapter`, as the slides are purely `ImageView` widgets:

```
package com.commonsware.android.preso.decktastic;

import android.content.Context;
import android.net.Uri;
import android.support.v4.view.PagerAdapter;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import com.squareup.picasso.Picasso;

class SlidesAdapter extends PagerAdapter {
  private PresoContents preso;
  private Context ctxt;

  SlidesAdapter(Context ctxt, PresoContents preso) {
    this.ctxt=ctxt;
    this.preso=preso;
  }

  @Override
  public Object instantiateItem(ViewGroup container, int position) {
    ImageView page=new ImageView(ctxt);
    ViewGroup.LayoutParams p=
      new ViewGroup.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,
        ViewGroup.LayoutParams.MATCH_PARENT);

    container.addView(page, p);

    Picasso.with(ctxt).load(getSlideImageUri(position)).into(page);

    return(page);
  }

  @Override
  public void destroyItem(ViewGroup container, int position,
                          Object object) {
    container.removeView((View)object);
  }

  @Override
  public int getCount() {
```

```
    return(preso.slides.size());
  }

  @Override
  public boolean isViewFromObject(View view, Object object) {
    return(view == object);
  }

  @Override
  public String getPageTitle(int position) {
    return(preso.getSlideTitle(position));
  }

  Uri getSlideImageUri(int position) {
    return(Uri.parse("file:///android_asset/"+preso.getSlideImage(position)));
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/SlidesAdapter.java)

Of note:

- `instantiateItem()` creates the `ImageView`, adds it to the supplied `container` (set to fill that container), and tells Picasso to go load the image asynchronously into the `ImageView`
- `destroyItem()` removes the `ImageView` from the container
- `getCount()` returns the number of pages, based on the number of slides in the `PresoContents` supplied to the `SlidesAdapter` via its constructor
- `getPageTitle()` returns the page title, obtained from the `PresoContents` object
- `getSlideImageUri()` gets a `Uri` pointing to a local file from the `PresoContents`, for use both by `instantiateItem()` and by the `Presentation` object that we will use for external display support (as will be covered later in this chapter)

Supporting the Direct-to-TV Scenario

To determine whether or not our activity is natively displaying on a TV-style screen, we check to see whether the device has either `FEATURE_TELEVISION` or `FEATURE_LEANBACK`, in the private `isDirectToTV()` method on `MainActivity`:

```
  private boolean isDirectToTV() {
    return(getPackageManager().hasSystemFeature(PackageManager.FEATURE_TELEVISION)
        || getPackageManager().hasSystemFeature(PackageManager.FEATURE_LEANBACK));
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Admittedly, not all Android direct-to-TV devices may advertise that they have one of these features. In particular, minor-brand Android HDMI sticks might just be using a fairly vanilla Android device profile, culled from a tablet. There is no good way of detecting such a scenario, though Decktastic could provide some manual option (e.g., checkable action item) to go into direct-to-TV mode if this proved to be important.

We use `isDirectToTV()` in two places. First, in `onCreate()`, we hide the action bar if we are going direct to a TV:

```
    if (isDirectToTV()) {
      getSupportActionBar().hide();
    }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Second, in `setupPager()`, we hide the `TabLayout` if we are going direct to a TV:

```
    if (!isDirectToTV()) {
      TabLayout tabs=(TabLayout)findViewById(R.id.tabs);

      tabs.setVisibility(View.VISIBLE);
      tabs.setupWithViewPager(pager);
      tabs.addOnTabSelectedListener(this);
    }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

This eliminates the "chrome" from our activity, leaving us with just the contents of the `ViewPager` itself, in the form of our slides. On the plus side, this gives us the visual output we want. However, it comes at a cost: there is no means for the presenter to change slides. After all, there is no touchscreen in this scenario, and so even though the `ViewPager` could be swiped, that is not possible without a touchscreen.

To support standard presentation remotes and similar mechanisms, `MainActivity` overrides `onKeyDown()`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  switch(keyCode) {
    case KeyEvent.KEYCODE_SPACE:
    case KeyEvent.KEYCODE_DPAD_RIGHT:
```

```
      case KeyEvent.KEYCODE_DPAD_DOWN:
      case KeyEvent.KEYCODE_PAGE_DOWN:
      case KeyEvent.KEYCODE_MEDIA_NEXT:
        if (pager.canScrollHorizontally(1)) {
          pager.setCurrentItem(pager.getCurrentItem()+1, true);
        }

        return(true);

      case KeyEvent.KEYCODE_DPAD_LEFT:
      case KeyEvent.KEYCODE_DPAD_UP:
      case KeyEvent.KEYCODE_PAGE_UP:
      case KeyEvent.KEYCODE_MEDIA_PREVIOUS:
        if (pager.canScrollHorizontally(-1)) {
          pager.setCurrentItem(pager.getCurrentItem()-1, true);
        }

        return(true);
    }

    return(super.onKeyDown(keyCode, event));
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Here, we will advance to the next slide if the user presses:

- the space bar or the Page Down key on a QWERTY keyboard
- right or down arrow keys, D-pad buttons, or the like
- a “next” media button on a media remote

Conversely, we will return to the preceding slide if the user presses:

- the Page Up key on a QWERTY keyboard
- left or up arrow keys, D-pad buttons, or the like
- a “previous” media button on a media remote

This should allow most remotes for direct-to-TV devices to control our slides. Note that we are passing `true` as the second parameter to the `setCurrentItem()` method, and therefore the audience will see an animated transition to the next slide. That may or may not be desirable; an enhanced edition of Decktastic might allow that to be configured (e.g., via a checkable action item).

Note that this is still a bit limited compared to having touchscreen access, as our `onKeyDown()` method only moves a slide at a time. There is no facility to jump to an

arbitrary spot, the way you could by swiping and tapping upon `ViewPager` tabs on a touchscreen.

Supporting External Displays

As noted earlier, in `onCreate()` of `MainActivity`, we create an instance of `PresentationHelper`, supplying the activity itself as both the `Context` and the `PresentationHelper.Listener` for presentation-related events:

```
    helper=new PresentationHelper(this, this);
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

That, in turn, requires us to forward along `onPause()` and `onResume()` events from our activity to the `PresentationHelper`:

```
@Override
public void onResume() {
  super.onResume();
  helper.onResume();
}

@Override
public void onPause() {
  helper.onPause();
  super.onPause();
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

We also have to implement `showPreso()` and `clearPreso()` methods to satisfy the `PresentationHelper.Listener` interface:

```
@Override
public void clearPreso(boolean showInline) {
  if (presoFrag != null) {
    presoFrag.dismiss();
    presoFrag=null;
  }
}

@Override
public void showPreso(Display display) {
  Uri slide=adapter.getSlideImageUri(pager.getCurrentItem());

  presoFrag=
```

```
        SlidePresentationFragment.newInstance(this, display, slide);
    presoFrag.show(getSupportFragmentManager(), "presoFrag");
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

In `showPreso()`, we obtain the `Uri` for the current slide by calling the `getSlideImageUri()` method we conveniently implemented on the `SlidesAdapter`. Then, we create an instance of a `SlidePresentationFragment`, handing it the slide `Uri`, and we `show()` that fragment. We only `dismiss()` the fragment in `clearPreso()`.

The fragment itself is a `PresentationFragment`, with an `ImageView` as the fragment's UI, populated using Picasso, with the `Uri` being transferred from the `newInstance()` factory method to the fragment itself via the arguments `Bundle`:

```
package com.commonsware.android.preso.decktastic;

import android.content.Context;
import android.net.Uri;
import android.os.Bundle;
import android.view.Display;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import com.squareup.picasso.Picasso;

public class SlidePresentationFragment extends PresentationFragment {
  private static final String KEY_URI="u";
  private ImageView slide=null;

  public static SlidePresentationFragment newInstance(Context ctxt,
                                                      Display display,
                                                      Uri slideUri) {
    SlidePresentationFragment frag=new SlidePresentationFragment();

    frag.setDisplay(ctxt, display);

    Bundle b=new Bundle();

    b.putParcelable(KEY_URI, slideUri);
    frag.setArguments(b);

    return(frag);
  }
```

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    slide=new ImageView(getContext());

    setSlideContent((Uri)getArguments().getParcelable(KEY_URI));

    return(slide);
  }

  void setSlideContent(Uri slideUri) {
    Picasso.with(getContext()).load(slideUri).into(slide);
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/SlidePresentationFragment.java)

While the `cwac-presentation` library contains a `PresentationFragment`, it is set up for the native API Level 11 implementation of fragments. Hence, Decktastic contains its own `PresentationFragment`, cloned from the `cwac-presentation` implementation, that uses the fragment backport, for use with our `AppCompatActivity`:

```
package com.commonsware.android.preso.decktastic;

import android.app.Dialog;
import android.app.Presentation;
import android.content.Context;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.view.Display;

abstract public class PresentationFragment extends DialogFragment {
  private Display display=null;
  private Presentation preso=null;

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    if (preso == null) {
      return(super.onCreateDialog(savedInstanceState));
    }

    return(preso);
  }
```

```
  public void setDisplay(Context ctxt, Display display) {
    if (display == null) {
      preso=null;
    }
    else {
      preso=new Presentation(ctxt, display, getTheme());
    }

    this.display=display;
  }

  public Display getDisplay() {
    return(display);
  }

  @Override
  public Context getContext() {
    if (preso != null) {
      return(preso.getContext());
    }

    return(getActivity());
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/PresentationFragment.java)

This arranges to show the *current* slide, for whatever the current slide is at the time `showPreso()` is called on `MainActivity`. However, we need to update this fragment to reflect changes in the current slide. To accomplish this, we set up `MainActivity` to implement the `OnTabSelectedListener` interface, then call `addOnTabSelectedListener()` on the `TabLayout` in `setupPager()` to have it forward tab-change events to the activity. Of those events, we pay particular attention to `onTabSelected()`, updating the `SlidePresentationFragment` if there is one around:

```
  @Override
  public void onTabReselected(TabLayout.Tab tab) {
    // unused
  }

  @Override
  public void onTabUnselected(TabLayout.Tab tab) {
    // unused
  }

  @Override
  public void onTabSelected(TabLayout.Tab tab) {
```

```
    if (presoFrag != null) {
      presoFrag
        .setSlideContent(adapter.getSlideImageUri(tab.getPosition()));
    }

    if (client!=null) {
      String url=preso.getSlideURL(tab.getPosition());

      client.play(Uri.parse(url), "image/png", null, 0, null, playCB);
    }
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

We will get into the `client` stuff from `onTabSelected()` in the next section, as that pertains to supporting remote playback devices.

Supporting Chromecast and Remote Playback Devices

The key limitation of Chromecast and other remote playback devices is that they can only play back media that they can access. While Chromecast supports mirroring, that is handled via the `Presentation` API discussed previously; devices limited to the `RemotePlaybackClient` API need URLs to media files. That is why our JSON for the presentation contains a URL pointing to a copy of each slide up on some public Web server. To push those URLs over to the Chromecast at the appropriate points, we need to set up the `RemotePlaybackClient` system.

First, in `onCreate()`, we define a `MediaRouteSelector` for remote playback devices, set up a `MediaRouter`, and add a callback to find out about selected route changes, asking `MediaRouter` to scan for possible routes along the way:

```
    selector=
        new MediaRouteSelector.Builder()
            .addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)
            .build();
    router=MediaRouter.getInstance(this);
    router.addCallback(selector, routeCB,
        MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

All of this is using the `mediarouter-v7` portion of the Android Support package, as the native `MediaRouter` and kin do not support remote playback devices.

Our menu resource for our action bar contains, among other things, a `MediaRouteActionProvider` from `mediarouter-v7`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:tools="http://schemas.android.com/tools"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/countdown"
    app:actionViewClass="com.commonsware.android.preso.decktastic.ReverseChronometer"
    app:showAsAction="always"
    tools:ignore="AlwaysShowAction,MenuTitle" />
  <item
    android:id="@+id/route_provider"
    android:title="@string/media_route_provider"
    app:actionProviderClass="android.support.v7.app.MediaRouteActionProvider"
    app:showAsAction="always" />
  <item
    android:id="@+id/first"
    android:icon="@android:drawable/ic_media_previous"
    android:title="@string/first"
    app:showAsAction="ifRoom" />
  <item
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:title="@string/last"
    app:showAsAction="ifRoom" />
  <item
    android:id="@+id/present"
    android:checkable="true"
    android:checked="true"
    android:title="@string/show_presentation"
    app:showAsAction="never" />

</menu>
```

(from Presentation/Decktastic/app/src/main/res/menu/activity_actions.xml)

As part of our work in setting up the action bar in `onCreateOptionsMenu()`, we retrieve the `MediaRouteActionProvider` and configure it with the same `MediaRouteSelector` that we used for the `MediaRouter` callback:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.activity_actions, menu);

    rc=(ReverseChronometer)menu.findItem(R.id.countdown)
                               .getActionView();

    rc.setWarningDuration(5 * 60);
    rc.setOnClickListener(this);
    rc.setOnLongClickListener(this);
    rc.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);
```

```
    rc.setTextColor(Color.WHITE);

    if (durationInSeconds>0) {
      rc.setOverallDuration(durationInSeconds);
    }

    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

    provider.setRouteSelector(selector);

    return(super.onCreateOptionsMenu(menu));
  }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

(the lines in `onCreateOptionsMenu()` pertaining to the `ReverseChronometer` will be explained later in this chapter)

If the user interacts with the `MediaRouteActionProvider` and elects to connect to a remote playback device, our `MediaRouter.Callback` will be notified about the change of route:

```
  private MediaRouter.Callback routeCB=new MediaRouter.Callback() {
    @Override
    public void onRouteSelected(MediaRouter router,
                                MediaRouter.RouteInfo route) {
      connect(route);
    }

    @Override
    public void onRouteUnselected(MediaRouter router,
                                  MediaRouter.RouteInfo route) {
      disconnect();
    }
  };
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Here, we just delegate the `onRouteSelected()` and `onRouteUnselected()` callbacks to `connect()` and `disconnect()` methods on `MainActivity`. `MediaRouter.Callback` is an abstract class, not an interface — otherwise, we would simply have implemented the interface on `MainActivity` and bypassed this anonymous inner class instance.

The `connect()` method on `MainActivity` is responsible for sending over the current slide to the remote playback device:

```
private void connect(MediaRouter.RouteInfo route) {
  client=
      new RemotePlaybackClient(getApplicationContext(), route);

  if (client.isRemotePlaybackSupported()) {
    String url=preso.getSlideURL(pager.getCurrentItem());

    client.play(Uri.parse(url), "image/png", null, 0, null, playCB);
  }
  else {
    client=null;
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Here, we:

- Create an instance of `RemotePlaybackClient`
- Confirm that the remote playback device supports the remote playback protocol (`isRemotePlaybackSupported()`)
- Call `play()`, passing over a URL pointing to the same slide that the `ViewPager` is showing from a local file

The `play()` call requires an `ItemActionCallback` as the last parameter. We really do not need the callback, but passing `null` does not work. So, we have a do-nothing `ItemActionCallback` named `playCB` that we use:

```
RemotePlaybackClient.ItemActionCallback playCB=
  new RemotePlaybackClient.ItemActionCallback() {
  @Override
  public void onResult(Bundle data, String sessionId,
                       MediaSessionStatus sessionStatus,
                       String itemId, MediaItemStatus itemStatus) {
  }

  @Override
  public void onError(String error, int code, Bundle data) {
  }
};
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

That will show the slide that was current as of the time the user connected to the remote playback device. We need to show a new slide when the presenter switches to a new slide, just as we did in the `Presentation` scenario. This too is handled in `onTabSelected()`, where we make the same sort of `play()` call that we did in `connect()`:

```
@Override
public void onTabSelected(TabLayout.Tab tab) {
  if (presoFrag != null) {
    presoFrag
      .setSlideContent(adapter.getSlideImageUri(tab.getPosition()));
  }

  if (client!=null) {
    String url=preso.getSlideURL(tab.getPosition());

    client.play(Uri.parse(url), "image/png", null, 0, null, playCB);
  }
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Not only is `disconnect()` called from `MediaRouter.Callback`, but it is also called from `onDestroy()` of `MainActivity`, where we also remove that callback from the `MediaRouter`:

```
@Override
public void onDestroy() {
  disconnect();
  router.removeCallback(routeCB);
  super.onDestroy();
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

`disconnect()` releases the `RemotePlaybackClient` and ensures that we are back on our default route:

```
private void disconnect() {
  if (client != null) {
    client.release();
    client=null;
  }

  router.getDefaultRoute().select();
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

The net effect of all of this is that the slides will update on the remote playback device as the presenter switches slides, in addition to when the presenter connects to the remote playback device originally. We are not in control of any transition effects — we simply provide the slides, and it is up to the remote playback device to download and show them, however that device wishes.

The Rest of the Story

One common need of a presenter is to know how much time is remaining in which to deliver the presentation. Presentations are usually time-limited, to fit conference agendas and the like. The JSON structure for a presentation contains the duration of the presentation, and it would be useful to let the presenter know how much of that duration is remaining.

The chapter on custom views has a section outlining the implementation of a `ReverseChronometer` widget. `Chronometer` is a standard Android SDK class for counting up time (e.g., a stopwatch). `ReverseChronometer` is for counting *down* time.

Decktastic puts a `ReverseChronometer` in the action bar as a custom view, courtesy of our menu XML:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:tools="http://schemas.android.com/tools"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/countdown"
    app:actionViewClass="com.commonsware.android.preso.decktastic.ReverseChronometer"
    app:showAsAction="always"
    tools:ignore="AlwaysShowAction,MenuTitle" />
  <item
    android:id="@+id/route_provider"
    android:title="@string/media_route_provider"
    app:actionProviderClass="android.support.v7.app.MediaRouteActionProvider"
    app:showAsAction="always" />
  <item
    android:id="@+id/first"
    android:icon="@android:drawable/ic_media_previous"
    android:title="@string/first"
    app:showAsAction="ifRoom" />
  <item
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:title="@string/last"
    app:showAsAction="ifRoom" />
```

```
  <item
    android:id="@+id/present"
    android:checkable="true"
    android:checked="true"
    android:title="@string/show_presentation"
    app:showAsAction="never" />

</menu>
```

(from Presentation/Decktastic/app/src/main/res/menu/activity_actions.xml)

We customize the `ReverseChronometer` through a handful of lines in the `onCreateOptionsMenu()` method:

```
    rc=(ReverseChronometer)menu.findItem(R.id.countdown)
                               .getActionView();

    rc.setWarningDuration(5 * 60);
    rc.setOnClickListener(this);
    rc.setOnLongClickListener(this);
    rc.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);
    rc.setTextColor(Color.WHITE);

    if (durationInSeconds>0) {
      rc.setOverallDuration(durationInSeconds);
    }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Here we:

- Retrieve the `ReverseChronometer` from the action item
- Have it change to a "warning" presentation with five minutes remaining
- Set up the activity to respond to click and long-click events
- Set the appearance to be `24sp` white text

And, if we already know the presentation's overall duration, via the `durationInSeconds` data member, we pour that into the `ReverseChronometer` as well.

`durationInSeconds` is populated via a few lines at the top of `setupPager()`:

```
  private void setupPager() {
    durationInSeconds=preso.duration * 60;

    if (rc!=null) {
      rc.setOverallDuration(durationInSeconds);
    }
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

This way, no matter whether `setupPager()` or `onCreateOptionsMenu()` is called first, we pour the duration into the `ReverseChronometer`.

By default, that `ReverseChronometer` does nothing other than show the remaining time... which remains fixed by default. That is where the click and long-click event handlers come into play:

```
@Override
public void onClick(View v) {
  ReverseChronometer rc=(ReverseChronometer)v;

  if (rc.isRunning()) {
    rc.stop();
  }
  else {
    if (isFirstRCClick) {
      isFirstRCClick=false;
      rc.reset();
    }

    rc.run();
  }
}

@Override
public boolean onLongClick(View v) {
  ReverseChronometer rc=(ReverseChronometer)v;

  rc.reset();

  return(true);
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

There are three possibilities when the user taps on the `ReverseChronometer`:

- It was never clicked before (`isFirstRCClick` is `true`), in which case we ensure that the `ReverseChronometer` is reset to the overall duration before calling `run()` to start the countdown
- It is already running, in which case we call `stop()` to pause the countdown
- It was not already running (but was clicked before), in which case we call `run()` again to resume the countdown

This gives us what from a media standpoint would be play, pause, and resume logic.

A long-click will `reset()` the `ReverseChronometer`, returning the time remaining to the overall duration.

Our action bar also has a few other action items, handled in `onOptionsItemSelected()`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.present:
      boolean original=item.isChecked();

      item.setChecked(!original);

      if (original) {
        helper.disable();
      }
      else {
        helper.enable();
      }

      break;

    case R.id.first:
      pager.setCurrentItem(0);
      break;

    case R.id.last:
      pager.setCurrentItem(adapter.getCount() - 1);
      break;
  }

  return(super.onOptionsItemSelected(item));
}
```

(from Presentation/Decktastic/app/src/main/java/com/commonsware/android/preso/decktastic/MainActivity.java)

Specifically:

- There is a checkable action item to determine whether or not we should be showing a `Presentation`. If this is unchecked, and an external display is attached, we still disable the `PresentationHelper`. This will cause normal display mirroring to begin, and the audience will see the same UI that the presenter does, complete with the `ViewPager`, action bar, and so on.

Checking it re-enables the `PresentationHelper`, so if an external display is available, we start showing the slides again.

- The `first` and `last` action bar items are "fast-forward" and "rewind" options, allowing the presenter to quickly jump to the first or the last slide in the presentation. This happens via calls to `setCurrentItem()` on the `ViewPager`, which will in turn invoke `onPageSelected()`, causing us to update our `PresentationFragment` or remote playback device, if needed.

Note that since the direct-to-TV mode hides the action bar, none of these options are available to the presenter on a device like Android TV or a Fire TV. This will require the presenter to use something else to track the remaining time in a presentation, such as a countdown timer app running on a separate Android device.

Creating a MediaRouteProvider

As was noted earlier in the book, you can use `MediaRouter` to identify media routes, such as those published by devices like Google's Chromecast. Specifically, remote playback routes let you write apps that tell other devices, like the Chromecast, to play back media on your behalf.

However, not only can you write *clients* for remote playback routes, you can write *providers* of those routes. Perhaps you are working with a hardware manufacturer that is creating a Chromecast-like device. Perhaps you want to allow your app, running on a Fire TV or an Android HDMI stick, to be controlled by a user's phone or tablet. Or perhaps you are trying to tie Android into specialized media hardware that does not communicate by conventional means (e.g., wireless speakers that do not use normal Bluetooth profiles).

This chapter will outline how you can create code that will publish media routes to users of `MediaRouter`, so that you can then take those requests and forward them to a remote device.

Prerequisites

This chapter assumes that you have read the chapter on `MediaRouter`.

Terminology

For the purposes of this chapter:

- The "client device" refers to a phone or tablet that runs an app that should be able to direct what is shown on a streaming media player

- The “player device” refers to the streaming media player itself, which may or may not be running Android
- The “player app” refers to an Android app running on an Android-powered player device

DIY Chromecast

Google’s Chromecast is a nice little device. However, it has issues:

- The device itself is not especially open.
- The Cast SDK that Google encourages for writing Chromecast-enabled apps is not especially open.
- The terms and conditions for using the Cast SDK may be troublesome for many developers.
- Chromecast is not available globally.
- Chromecast is only one device, and there are plenty of other streaming media devices available that need to be considered.

Some of these issues can be mitigated by the use of `MediaRouter` and `RemotePlaybackClient` instead of the proprietary Cast SDK. You are not bound by any particular license terms (beyond the norm for Android development) and the implementation of the media framework is open.

However, to make this work, the client device needs to know how to talk to the player device.

The good news is that the media routing framework in Android supports plug-in media route providers for just this purpose. The OS ships with such a provider for the Chromecast, and you can create your own providers to talk to whatever else you would like to talk to. The user can then install a small app on their client device that implements this media route provider, and *any* apps already on their client device that use classes like `RemotePlaybackClient` will automatically be able to cast their desired content to the player device.

MediaRouteProvider

The guts of this come in the form of a `MediaRouteProvider`. Your custom subclass of `MediaRouteProvider` will:

- Tell Android what general capabilities you support, such as remote playback, session management, and the like
- Advertise what sorts of content your player device is capable of playing (e.g., certain video MIME types, certain URL schemes like `http` and `rtsp`)
- Serve as the recipient of commands from `MediaRouter`, `RemotePlaybackClient`, and the like, for you to forward along asynchronously to the player device

Depending upon your use case, you could elect to keep the `MediaRouteProvider` private to your application. That way, your app can cast to the player device, but no other apps can. Or, you can make your `MediaRouteProvider` available to all apps on the device, with the media routing framework taking care of the IPC details to have those apps tell your `MediaRouteProvider` what the player device should do.

Player Device… and Maybe a Player App

Of course, this assumes the existence of some player device that is not supported by Android out of the box. Since Android only really supports Chromecast, external displays (e.g., HDMI, MHL, Miracast), and some Bluetooth options (e.g., external speakers) for media routes, there are countless player devices that need additional help. These will run the gamut from devices from major players (e.g., Amazon's Fire TV) to no-name devices (e.g., Android HDMI "sticks").

Some player devices will run Android. In that case, you would be writing a player app that would run on the player device that would be the recipient of commands sent to it from your `MediaRouteProvider` on the client device. For example, if you write a video player app, you could augment it with remote control capability driven by a `MediaRouteProvider` on a client device, turning your player app and anything it can run on (e.g., Fire TV, OUYA game console) into a Chromecast-like environment.

Some player devices will not run Android. If they offer some existing remote control over-the-air protocol, you could create a `MediaRouteProvider` that speaks that protocol. Or, perhaps the player devices are programmable, just not via Android (e.g., a Linux program for XMBC), in which case you might be able to write both ends of the communications channel.

Communications Protocol

Somehow, the data from the `MediaRouteProvider` needs to get to the player device (and, where relevant, the player app). Likely candidates include Bluetooth, regular WiFi (if both devices are on the same network), and WiFi Direct.

However, in principle, anything is possible. For example, there is nothing stopping you from sending `MediaRouteProvider` commands to some Web server out on the Internet, which forwards them to some distant location for use. That would be a bit unusual – normally, the user of the client device is controlling something she can see — but it certainly could be done.

The biggest thing to watch out for is the addressability of the media to be played back. There is little point in connecting a `MediaRouteProvider` to some player device, then not have the ability for the player device to access the media that the client device is requesting. The expected pattern is that the media is hosted in some (relatively) central location, like a Web server. However, once again, anything is possible. If you want to have some sort of server on the client device, to allow the player device to play back media from it, and you believe that you can adequate secure this, you are welcome to do so.

Creating the MediaRouteProvider

As noted earlier, the core of all of this is a custom `MediaRouteProvider`. Google supplies a sample application for creating such a `MediaRouteProvider`. However, it is overly complex, and it is undocumented.

This chapter will focus instead on the MediaRouter/RouteProvider sample project. This is a clone of the `MediaRouter/RemotePlayback` sample project covered earlier in this book, with the addition of a custom `MediaRouteProvider`.

Defining the Supported Actions

A `MediaRouteProvider` advertises — whether to its own app's `MediaRouter` or to the entire device — what sorts of actions it can perform. For example, a remote playback route provider needs to support actions like play, pause, resume, and stop of some piece of media.

The way this is handled in the media routing framework is via a series of `IntentFilter` objects.

Since `IntentFilter` objects do not need a `Context` to be created, it is safe to define them statically, if desired. That's what we do in `DemoRouteProvider`, a custom subclass of `MediaRouteProvider`. It declares a pair of `static final IntentFilter` objects, `ifPlay` and `ifControl`, which are then configured in a `static` initialization block:

```
private static final IntentFilter ifPlay=new IntentFilter();
private static final IntentFilter ifControl=new IntentFilter();

static {
  ifPlay.addCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK);
  ifPlay.addAction(MediaControlIntent.ACTION_PLAY);
  ifPlay.addDataScheme("http");
  ifPlay.addDataScheme("https");
  ifPlay.addDataScheme("rtsp");

  try {
    ifPlay.addDataType("video/*");
  }
  catch (MalformedMimeTypeException e) {
    throw new RuntimeException("Exception setting MIME type", e);
  }

  ifControl.addCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK);
  ifControl.addAction(MediaControlIntent.ACTION_PAUSE);
  ifControl.addAction(MediaControlIntent.ACTION_RESUME);
  ifControl.addAction(MediaControlIntent.ACTION_STOP);
  ifControl.addAction(MediaControlIntent.ACTION_GET_STATUS);
  ifControl.addAction(MediaControlIntent.ACTION_SEEK);
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteProvider.java)

Both stipulate that they are looking for `Intent` objects in the `MediaControlIntent.CATEGORY_REMOTE_PLAYBACK` category. This category is used for all media routing `Intents` that form the foundation of the routing framework.

`ifPlay` is defined as supporting `MediaControlIntent.ACTION_PLAY`, stating that we know how to play back some content. The qualifications for "some content" are handled via scheme and type constraints placed on the `IntentFilter`. Here, we limit the content to be URLs that might be reachable by a playback device (`http`, `https`, `rtsp`) and have a MIME type matching `video/*`. Hence, we are stating that we can play back streaming video.

`ifControl` sets up the remaining actions that we support:

- `MediaControlIntent.ACTION_PAUSE`
- `MediaControlIntent.ACTION_RESUME`
- `MediaControlIntent.ACTION_STOP`
- `MediaControlIntent.ACTION_GET_STATUS`
- `MediaControlIntent.ACTION_SEEK`

These are placed on an independent `IntentFilter` because, technically, we can support these actions on any type of media. In the case of this specific example, the only media we support is streaming video. But, we could configure other `IntentFilter` objects, like `ifPlay` was, stating yet other media types that we handle.

To fully comply with the `RemotePlaybackClient` API, we must advertise that we handle all of those actions... even if our intended client will not use all of them.

We could also:

- Advertise that we support session management actions, like `MediaControlIntent.ACTION_START_SESSION`
- Advertise that we support the "enqueue" operation for stacking up media to be played (e.g., `MediaControlIntent.ACTION_ENQUEUE`) and manipulating that queue (e.g., `MediaControlIntent.ACTION_REMOVE`)
- Define a custom category for other actions that we support that are "out of band" with respect to the standard media routing actions

All of those are demonstrated in Google's sample app.

Creating the Descriptors

Just because we have some `static IntentFilter` objects does not mean that anything will pay attention to them. We need to actually register them with the media routing framework, wrapped in a pair of "descriptor" objects. `DemoRouteProvider` calls a private `handleDiscovery()` method from the constructor, where `handleDiscovery()` sets up the descriptors:

```
private void handleDiscovery() {
  MediaRouteDescriptor.Builder mrdBuilder=
      new MediaRouteDescriptor.Builder(DEMO_ROUTE_ID, "Demo Route");

  mrdBuilder.setDescription("The description of a demo route")
            .addControlFilter(ifPlay)
            .addControlFilter(ifControl)
            .setPlaybackStream(AudioManager.STREAM_MUSIC)
```

```
            .setPlaybackType(MediaRouter.RouteInfo.PLAYBACK_TYPE_REMOTE)
            .setVolumeHandling(MediaRouter.RouteInfo.PLAYBACK_VOLUME_FIXED);

    MediaRouteProviderDescriptor.Builder mrpdBuilder=
        new MediaRouteProviderDescriptor.Builder();

    mrpdBuilder.addRoute(mrdBuilder.build());

    setDescriptor(mrpdBuilder.build());
  }
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteProvider.java)

In the end, we need to provide a `MediaRouteProviderDescriptor` to the `MediaRouteProvider` by means of a `setDescriptor()` method. `MediaRouteProviderDescriptor` is, in effect, metadata about the `MediaRouteProvider` itself. At the present time, the only thing this holds is a set of `MediaRouteDescriptor` objects, one for each media route that the `MediaRouteProvider` claims to support.

A `MediaRouteProvider` is made up of several pieces of information, including:

- The `IntentFilter`(s) representing the supported actions and, where relevant, MIME types and schemes
- A locally-unique ID of the route, to distinguish it from any other one that we might configure
- A name and description, which the user will see when they try to connect to this route (e.g., via a `MediaRouteActionProvider`)
- What audio stream is being used for the playback, from the standpoint of volume management, audio ducking, and the like
- Whether the playback is occurring locally on the device to some peripheral (e.g., speaker) or if the playback is occurring remotely on a player device (e.g., Chromecast)
- Whether playback volume is controlled here on the client device or on the player device
- Etc.

These are all configured on a `MediaRouteProvider` by creating a `MediaRouteProvider.Builder` and supplying the values either in the `Builder` constructor or via fluent setter methods. In the particular case of our simple demo provider, we:

- Use various strings for the ID, name, and description

- Use the two `IntentFilter` objects defined earlier to indicate what actions we can perform
- Indicate that the playback stream is `STREAM_MUSIC`, that the playback type is `PLAYBACK_TYPE_REMOTE`, and that the volume handling is `PLAYBACK_VOLUME_FIXED` (i.e., volume should be managed on the TV or whatever the media is being played upon)

It is very likely that you will elect to have several `MediaRouteDescriptor` objects for different client application scenarios. Google's sample app uses a total of four `MediaRouteDescriptor` objects:

- One set up largely like the one in this sample
- One set up with `PLAYBACK_VOLUME_VARIABLE` (so volume is controllable by a client app)
- One set up with variable volume plus queuing actions
- One set up with variable volume plus queuing and session management actions

Receiving the Actions

Now, we have told the media routing framework what actions we support. Some app will then try to use `RemotePlaybackClient` and ask us to perform those actions. Hence, we need to find out when this happens, so we can do the actual work of having the playback device actually play back the media, pause the media, etc.

To do this, we need to create a custom subclass of `MediaRouteProvider.RouteController`. This contains a series of callback methods which we can override to find out when various events occur.

There are four such callback methods that the `DemoRouteController` subclass of `MediaRouteProvider.RouteController` implements:

- `onSelect()`, which will be called when a client app has selected our `MediaRouteProvider` to handle some media on behalf of that client app
- `onUnselect()` and `onRelease()`, which will be called when the client app disconnects from our `MediaRouteProvider`
- `onControlRequest()`, which will be called when some specific action that we advertised is requested, such as playing back a piece of media

The `DemoRouteController` just logs a message to Logcat for the first three callbacks:

```
@Override
public void onRelease() {
  Log.d(getClass().getSimpleName(), "released");
}

@Override
public void onSelect() {
  Log.d(getClass().getSimpleName(), "selected");
}

@Override
public void onUnselect() {
  Log.d(getClass().getSimpleName(), "unselected");
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteController.java)

The `onControlRequest()` method is a bit more complex, as *all* control requests route through here: play, pause, resume, stop, etc. `onControlRequest()` is passed the `Intent` identifying the particular action that should be performed, and we can examine the `Intent` action string to determine what needs to be done. In this case, `onControlRequest()` delegates the real work to action-specific methods like `onPlayRequest()`:

```
@Override
public boolean onControlRequest(Intent i, ControlRequestCallback cb) {
  if (i.hasCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)) {
    if (MediaControlIntent.ACTION_PLAY.equals(i.getAction())) {
      return(onPlayRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_PAUSE.equals(i.getAction())) {
      return(onPauseRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_RESUME.equals(i.getAction())) {
      return(onResumeRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_STOP.equals(i.getAction())) {
      return(onStopRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_GET_STATUS.equals(i.getAction())) {
      return(onGetStatusRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_SEEK.equals(i.getAction())) {
      return(onSeekRequest(i, cb));
    }
  }
```

```
    Log.w(getClass().getSimpleName(), "unexpected control request"
        + i.toString());

    return(false);
  }
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteController.java)

`onControlRequest()` should return `true` if we agree to perform the action and will use the supplied `ControlRequestCallback` object to asynchronously deliver our results. If `onControlRequest()` returns `false`, that means that we are rejecting the action for some reason, such as it being one that is unrecognized. In `DemoRouteController`, that will occur if the category or the action on the `Intent` is not one of the supported options.

Note that if you opted into variable volume, there are `onSetVolume()` and `onUpdateVolume()` callback methods that will give you access to those events.

Handling the Actions

For those actions that you advertise and receive in `onControlRequest()`, you need to actually do the work for those actions. The details of this will vary widely depending upon your playback device and playback app that you are supporting. For example, you might establish a WiFi Direct connection in `onSelect()`, then use that connection in handling play, pause, etc. actions.

However, a few aspects of handling these actions will be in common across all implementations:

- `onControlRequest()` must return `true` or `false` as was described in the preceding section
- You must call `onResult()` or `onError()` on the `ControlRequestCallback` object to indicate if the action succeeded or failed
- You must supply an appropriate `Bundle` to those methods, particularly to `onResult()`, containing the right set of values to provide more details about the results of the action

The details of what that `Bundle` must contain are documented on the `MediaControlIntent` class, on the definition of each action string (e.g., `ACTION_PLAY`).

With that in mind, let's look at the six actions supported by `DemoRouteController`.

Play

The `Bundle` passed to `onResult()` of the `ControlRequestCallback`, when the action is `ACTION_PLAY`, needs three values:

- `EXTRA_SESSION_ID`: if you are implementing session management, this will be the unique session ID (`String`) for the session you are playing the media in. If you are not implementing session management, then what you are supposed to return is undocumented and (hopefully) unused
- `EXTRA_ITEM_ID`: if you are implementing "enqueue" support, this will be the item ID (`String`) for managing this item in the queue of available items. If you are not supporting a playback queue, then what you are supposed to return is undocumented and (hopefully) unused
- `EXTRA_ITEM_STATUS`: this should point to a `Bundle` created from a `MediaItemStatus` object where you indicate what the status is of the playback of this item

You create a `MediaItemStatus` object via a `MediaItemStatus.Builder`, where you can pass into the constructor a value indicating the overall status (e.g., `MediaItemStatus.PLAYBACK_STATE_PLAYING`), plus use fluent setter methods to define additional characteristics of the status, such as the current seek position.

The `DemoRouteController` logic for `ACTION_PLAY`, in the `onPlayRequest()` method, logs the event to Logcat and crafts a valid-but-meaningless result `Bundle` for use with `onResult()`:

```
private boolean onPlayRequest(Intent i, ControlRequestCallback cb) {
  Log.d(getClass().getSimpleName(), "play: "
      + i.getData().toString());

  MediaItemStatus.Builder statusBuilder=
      new MediaItemStatus.Builder(
                                MediaItemStatus.PLAYBACK_STATE_PLAYING);

  Bundle b=new Bundle();

  b.putString(MediaControlIntent.EXTRA_SESSION_ID, DemoRouteProvider.DEMO_SESSION_ID);
  b.putString(MediaControlIntent.EXTRA_ITEM_ID, DemoRouteProvider.DEMO_ITEM_ID);
  b.putBundle(MediaControlIntent.EXTRA_ITEM_STATUS,
              statusBuilder.build().asBundle());

  cb.onResult(b);

  return(true);
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteController.java)

Pause, Resume, and Stop

The `Bundle` passed to `onResult()` of the `ControlRequestCallback`, when the action is `ACTION_PAUSE`, `ACTION_RESUME`, or `ACTION_STOP`, does not need any particular values at the present time. Hence, the `DemoRouteController` methods for those actions just log the event to Logcat and pass an empty `Bundle` to `onResult()`:

```
private boolean onPauseRequest(Intent i, ControlRequestCallback cb) {
  Log.d(getClass().getSimpleName(), "pause");

  cb.onResult(new Bundle());

  return(true);
}

private boolean onResumeRequest(Intent i, ControlRequestCallback cb) {
  Log.d(getClass().getSimpleName(), "resume");

  cb.onResult(new Bundle());

  return(true);
}

private boolean onStopRequest(Intent i, ControlRequestCallback cb) {
  Log.d(getClass().getSimpleName(), "stop");

  cb.onResult(new Bundle());

  return(true);
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteController.java)

Get Status and Seek

The `Bundle` passed to `onResult()` of the `ControlRequestCallback`, when the action is `ACTION_GET_STATUS` or `ACTION_SEEK`, must contain the same sort of `MediaItemStatus`-built nested `Bundle` representing the current status. For `ACTION_GET_STATUS`, the only "work" to be done is to pass back the status; for `ACTION_SEEK`, you should move the playback position to the location indicated by an extra on the `Intent`, then return the revised status.

In the case of `DemoRouteController`, both just log a message to Logcat and return a fairly pointless status:

```
private boolean onGetStatusRequest(Intent i,
                                   ControlRequestCallback cb) {
  Log.d(getClass().getSimpleName(), "get-status");

  MediaItemStatus.Builder statusBuilder=
      new MediaItemStatus.Builder(
                                   MediaItemStatus.PLAYBACK_STATE_PLAYING);

  Bundle b=new Bundle();

  b.putBundle(MediaControlIntent.EXTRA_ITEM_STATUS,
              statusBuilder.build().asBundle());

  cb.onResult(b);

  return(true);
}

private boolean onSeekRequest(Intent i, ControlRequestCallback cb) {
  Log.d(getClass().getSimpleName(), "seek");

  MediaItemStatus.Builder statusBuilder=
      new MediaItemStatus.Builder(
                                   MediaItemStatus.PLAYBACK_STATE_PLAYING);

  Bundle b=new Bundle();

  b.putBundle(MediaControlIntent.EXTRA_ITEM_STATUS,
              statusBuilder.build().asBundle());

  cb.onResult(b);

  return(true);
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteController.java)

Publishing the Controller

While we have defined our `RouteController`, we still need to teach our `MediaRouteProvider` about it. That is through overriding the `onCreateRouteController()` method and returning an instance of `RouteController`:

```
@Override
public RouteController onCreateRouteController(String routeId) {
```

```
    return(new DemoRouteController());
  }
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteProvider.java)

`onCreateRouteController()` is passed the route ID `String` used in the `MediaRouteDescriptor`. You can either use that to instantiate a different `RouteProvider`, pass the `String` into a common `RouteProvider` so it knows what to do, or ignore it entirely if you have only one published route. In the case of `DemoRouteProvider`, we ignore the route ID and always return a `DemoRouteController`.

Handling Discovery Requests

`DemoRouteProvider` is always available, largely because it does not do much of anything.

In the real world, your `MediaRouteProvider` may not always be relevant. For example, the TV you are set up to talk to may be powered down. Or, the user may not be at home where the TV is, so the client device and the TV are not on the same network.

Rather than constantly polling the outside world to see if a route is possible, we only do this when a client app requests "route discovery", such as by providing the `MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY` flag on an `addCallback()` call to a `MediaRouter`. That in turn triggers an `onDiscoveryRequestChanged()` call on our `MediaRouteProvider`.

There, and in our constructor-triggered setup, we should do work to determine if a route is currently possible and set up our descriptors. This work should be done in a background thread if it involves network I/O.

Note that `onDiscoveryRequestChanged()` is passed a `MediaRouteDiscoveryRequest` object, describing what the consuming app is looking for. If the request is irrelevant for your provider (e.g., the app wants a local audio route, and you provide remote playback routes), simply ignore it.

The `onDiscoveryRequestChanged()` implementation in `DemoRouteProvider` just calls the same `handleDiscovery()` method that the constructor does.

Consuming the MediaRouteProvider

Having a `MediaRouteProvider` is nice, but it is useless if apps are not going to know about it.

You have two main options for consuming the `MediaRouteProvider`: use it only within your own app, or publish it to all apps on the device.

Private Provider

Using a `MediaRouteProvider` for your own app is very simple. Just add a single call to `addProvider()` on your `MediaRouter`, supplying an instance of your `MediaRouteProvider`.

Since our sample project is a fork of the original `RemotePlaybackClient` sample, we still have a `PlaybackFragment` that sets up the `MediaRouter` and `MediaRouteActionProvider`. In `onAttach()` of that `PlaybackFragment`, we can configure our `MediaRouterProvider` after obtaining the `MediaRouter` instance:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);

  router=MediaRouter.getInstance(host);
  provider=new DemoRouteProvider(getActivity());
  router.addProvider(provider);
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/PlaybackFragment.java)

At this point, our `DemoRouteProvider` will be available as an option for the user, along with any other eligible media routes:

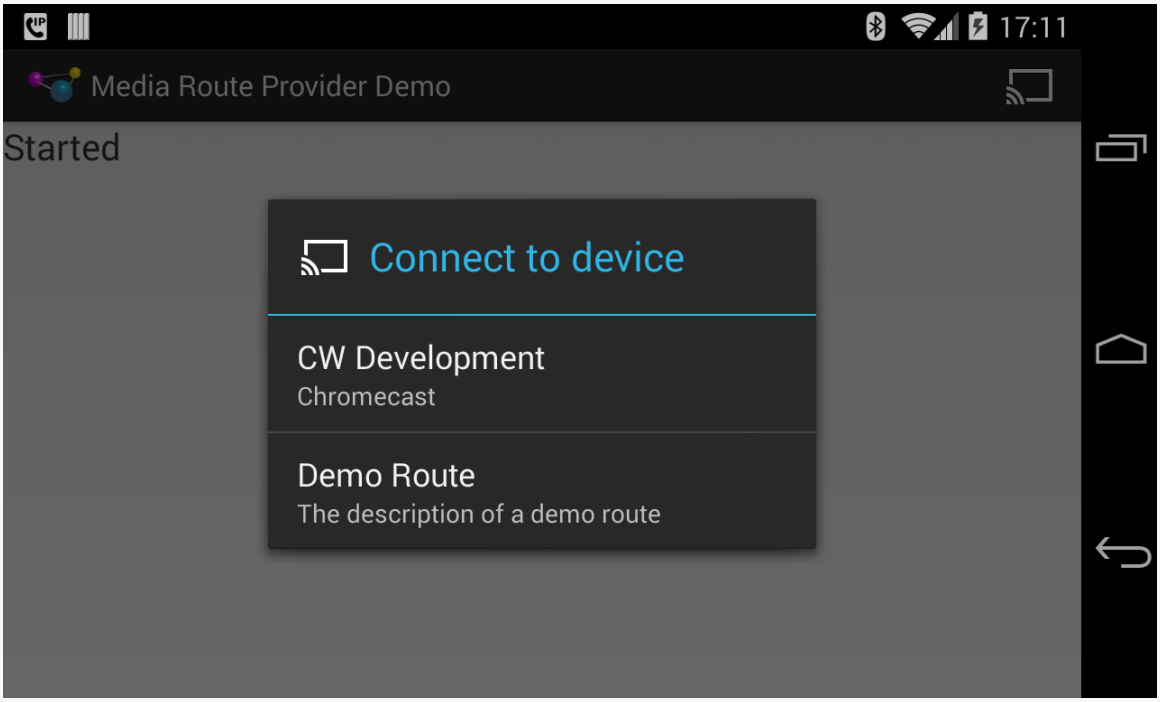


Figure 851: MediaRouteProvider Demo, on a Nexus 4, Showing Available Routes

Choosing the `DemoRouteProvider` ("Demo Route" in the screenshot) will allow you to use it just like you do a Chromecast... if you do not mind the fact that nothing shows up on your television:

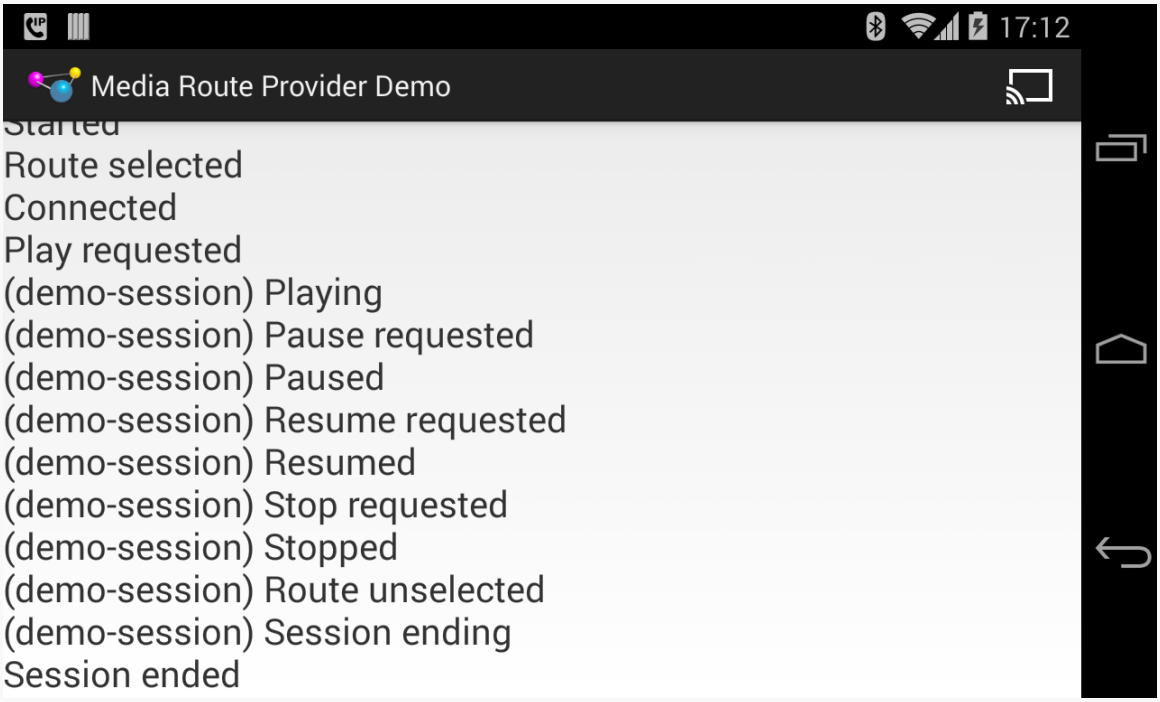


Figure 852: MediaRouteProvider Demo, on a Nexus 4, After Several Commands

As it turns out, the `DemoRouteProvider` works better than Google's own `MediaRouteProvider` for the Chromecast, insofar as more of the callbacks work. Specifically, we actually receive callbacks for pause, resume, and stop events, as opposed to having to just assume that those events completed.

Also, we remove the demo provider in `onDetach()`:

```
@Override
public void onDetach() {
  router.removeProvider(provider);

  super.onDetach();
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/PlaybackFragment.java)

Among other things, this allows us to correctly handle configuration changes — if we fail to call `removeProvider()` and blindly add another provider in `onAttach()`, we wind up with multiple providers, because our `MediaRouter` is a framework-provided singleton and is not re-created with the new fragment.

Public Provider

If you want your `MediaRouteProvider` to be used by other apps, you will need to create one more Java class: a subclass of `MediaRouteProviderService`. This requires only one method, `onCreateMediaRouteProvider()`, where you return an instance of your `MediaRouteProvider`:

```
package com.commonsware.android.mrp;

import android.support.v7.media.MediaRouteProvider;
import android.support.v7.media.MediaRouteProviderService;

public class DemoRouteProviderService extends MediaRouteProviderService {
  @Override
  public MediaRouteProvider onCreateMediaRouteProvider() {
    return(new DemoRouteProvider(this));
  }
}
```

(from MediaRouter/RouteProvider/app/src/main/java/com/commonsware/android/mrp/DemoRouteProviderService.java)

This also needs to be added to your manifest, like any other `Service`. Give it an `<intent-filter>` looking for the `android.media.MediaRouteProviderService` action, so the media routing framework knows that it can obtain a `MediaRouteProvider` from it:

```
<service
  android:name="DemoRouteProviderService"
  tools:ignore="ExportedService">
  <intent-filter>
    <action android:name="android.media.MediaRouteProviderService"/>
  </intent-filter>
</service>
```

However, do *not* do both `addProvider()` *and* have the `<service>` element. If you use the `<service>` element, your app can use the `MediaRouteProvider`, just as can any other app on the device. Hence, in the published source code for this sample, the `<service>` element is commented out — you will need to uncomment it, and comment out the `addProvider()` call, to test the `DemoRouteProvider` with other apps.

Implementing This "For Realz"

Of course, `DemoRouteProvider` is just a demo and does not actually play any media anywhere. It is here to give you the basic steps for responding to `RemotePlaybackClient` requests. For a production `MediaRouteProvider`, in addition to the usual tightening-up of the code (e.g., better exception handling), you will need to work on other areas as well, ones that are beyond the scope of the sample app.

Communicating with the Playback Device

Of course, the big one is passing the actions over to the playback device, so you actually *do* play back media.

If you are the developer of the playback device and its protocols (e.g., it is an Android device, and you are writing the playback app for it), then you can choose how you wish to handle the communications. You can work with low-level socket protocols directly, or you can leverage libraries like AllJoyn or ZeroMQ.

If the playback device "is what it is", and you cannot change it, then you will need to determine what protocols it offers and how best to map the `MediaControlIntent` actions to that protocol.

Also note that `onControlRequest()` is designed for asynchronous operation. The sample app just invoked the `ControlRequestCallback` during the `onControlRequest()` processing. Usually, though, your communications with the playback device will not be as fast as a call to `Log.d()`. You should arrange to do those communications in a background thread, perhaps via a single-thread thread pool as an `ExecutorService`. Simply pass the `ControlRequestCallback` to that thread along with the rest of the action's data (e.g., the URL of the media to load), and the thread can call `onResult()` or `onError()` as needed.

Handling Other Actions/Protocols

As was noted in the description of the sample app, that app avoids:

- volume control
- session management
- queue management

Any of those may be of interest to your users, and so you may need to consider offering them at some point. Also note that some potential client apps might need those capabilities and therefore will not see or use your published media routes without them.

Custom Actions

When setting up the `MediaRouteProvider`, we create one or more `MediaRouteDescriptor` objects wrapped around one or more `IntentFilter` objects. Those `IntentFilter` objects indicate what actions we support. The `DemoRouteProvider` uses standard actions (e.g., `ACTION_PLAY`) in a standard category (`CATEGORY_REMOTE_PLAYBACK`).

However, you are not limited to that.

You are welcome to also support custom actions in a custom category, to represent other things that your particular `MediaRouteProvider` offers. You can then use those actions from your own client app, or document them for use by third-party apps.

The client app can use `supportsControlRequest()` and `sendControlRequest()` to determine whether a particular media route supports a particular `Intent` that represents an action to be performed by that route's `MediaRouteProvider`. This way, a client app can work both with your custom `MediaRouteProvider` (taking advantage of your custom actions) and with regular providers that lack such support, assuming that the client can gracefully degrade its functionality.

Google's sample app defines a custom `ACTION_GET_STATISTICS` action that their sample client requests where available and their sample provider implements.

The Media Projection APIs

Android 5.0 debuted the ability for Android apps to take screenshots of whatever is in the foreground. It further allows apps to record full-resolution video of whatever is in the foreground, for screencasts, product demo videos, and the like. For whatever reason, this is called "media projection", and is based around classes like `MediaProjectionManager`.

In this chapter, we will explore how to use the media projection APIs to record screenshots and screencast-style videos.

Prerequisites

Understanding this chapter requires that you have read the core chapters, plus the chapter on embedding a Web server in your app for debug and diagnostic purposes.

Having read the chapter on using the camera APIs would not be a bad idea, particularly for video recording, though it is not essential.

Requesting Screenshots

Here, "screenshot" (or "screen capture") refers to generating an ordinary image file (e.g., PNG) of the contents of the screen. Most likely, you have created such screenshots yourself for a desktop OS (e.g., using the PrtSc key on Windows or Linux). Android's development tools allow you to take screenshots of devices and emulators, and there is a cumbersome way for users to take screenshots using the volume and power keys.

The media projection APIs allow you to take a screenshot of whatever is in the foreground... which does not necessarily have to be your own app. Indeed, you can take screenshots of any app, plus of system-supplied UI, such as the pull-down notification shade.

Not surprisingly, this has privacy and security issues. As such, in order to be able to take screenshots, the user must agree to allow it. In particular, instead of a durable permission that the user might grant once and forget about, the user has to agree to allow your app to take screenshots every time you want to do so.

Introducing andprojector

In 2009, the author of this book wrote a utility called DroidEx. This tool ran on a desktop or notebook and served as a "software projector" for an Android device, as opposed to the hardware projectors (e.g., ELMO) usually needed to show an Android screen to a large audience. Under the covers, DroidEx used the same protocol that Android Studio and DDMS use for screenshots, requesting screenshots as fast as possible, drawing them to a Swing `JFrame`. Later, Jens Riboe took DroidEx a bit further, adding more of a Swing control UI, in the form of Droid@Screen.

The `MediaProjection/andprojector` sample project has the same objective as did DroidEx: be able to show the contents of an Android screen to an audience. Nowadays, you might be able to do that straight from hardware, using things like an MHL->HDMI adapter. However, sometimes that option is not available (e.g., the projector you are using for your notebook is limited to VGA). andprojector differs from DroidEx in a few key ways:

- It is an Android app, not a program that you run on your notebook, and so it can be used without a notebook that has the Android SDK on it (which DroidEx required)
- It "projects" the screen using an embedded Web server to push PNG files to a Web browser, as opposed to DroidEx's use of a Swing `JFrame` to display the projection in a desktop OS window
- It uses the media projection APIs, which is the point of this chapter

On the device, the UI resembles that of the Web server apps profiled elsewhere in this book. When launched, the screen is mostly empty, except for a phone action bar item:

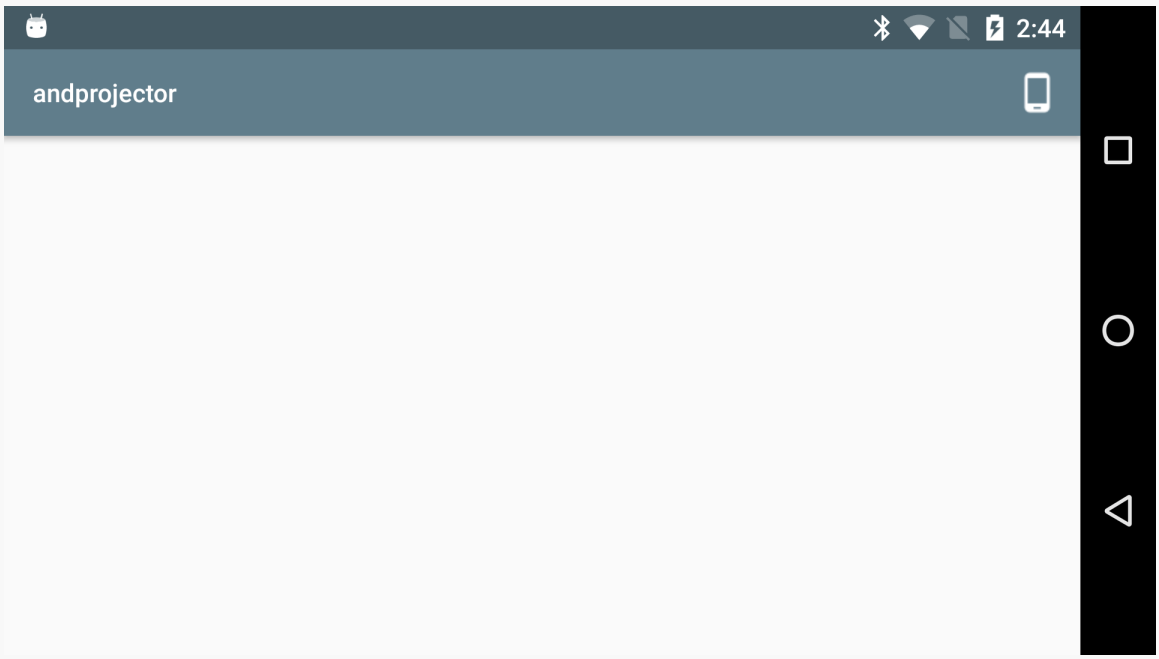


Figure 853: andprojector, As Initially Launched

When you tap the action bar item, a system-supplied dialog appears, asking for permission to take screenshots:

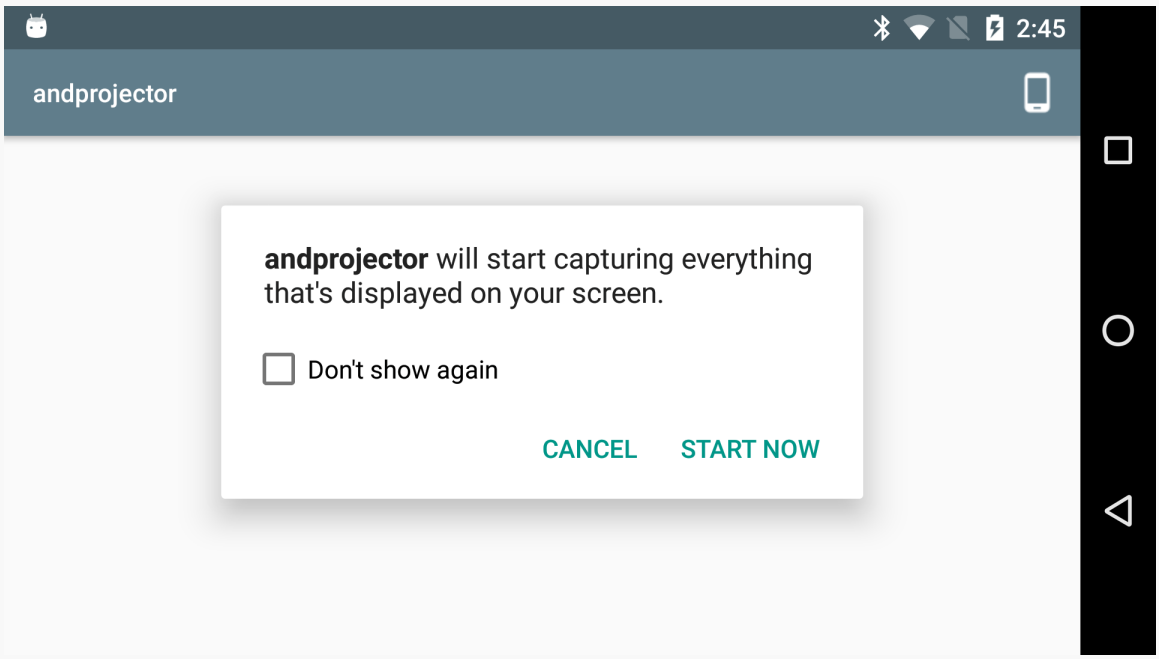


Figure 854: andprojector, Showing Permission Dialog

If you grant permission, you will see URLs that can be used to view what is on the device screen:

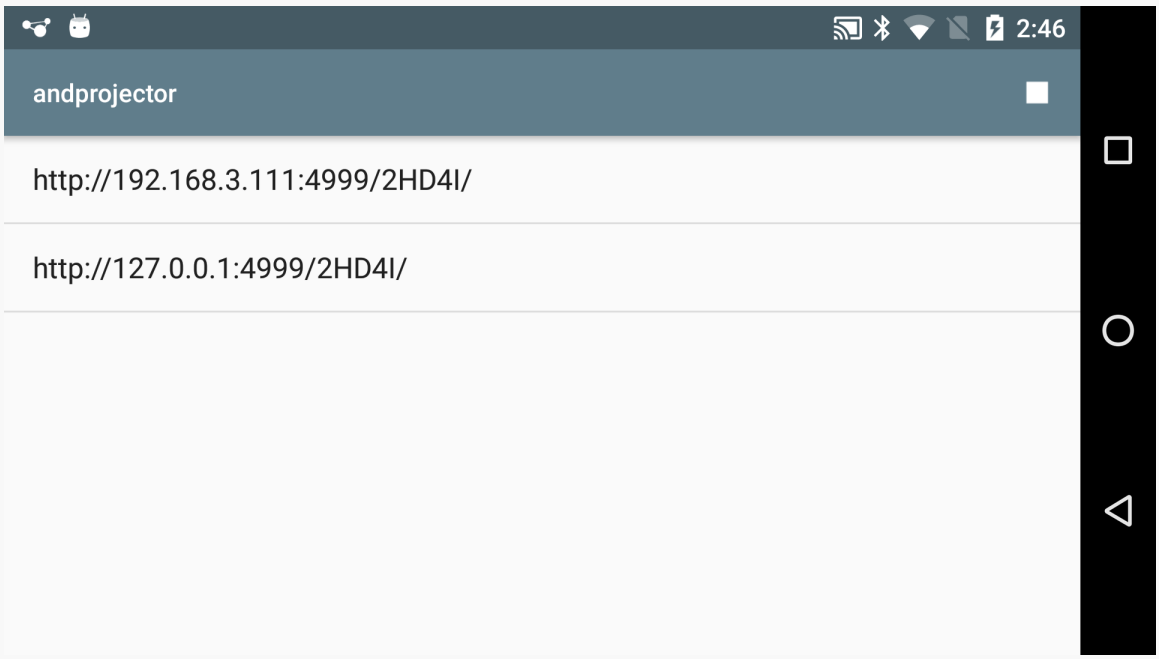


Figure 855: andprojector, Showing URLs

Entering one of those (including the trailing slash!) in a Web browser on some other machine on the same WiFi network will cause it to start showing the contents of the device screen. This can be done in either orientation, though it tends to work better in landscape.

Clicking the "stop" action bar item — which replaced the device action bar item when permission was granted — will stop the presentation and return the app to its original state.

With that in mind, let's see how andprojector pulls off this bit of magic.

Asking for Permission

In the `MainActivity` that houses our UI, in `onCreate()`, we get our hands on a `MediaProjectionManager` system service, in addition to fussing with Material-style coloring for the status bar:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  Window window=getWindow();
```

```
    window.addFlags(WindowManager.LayoutParams.FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS);
    window.clearFlags(WindowManager.LayoutParams.FLAG_TRANSLUCENT_STATUS);
    window.setStatusBarColor(
      getResources().getColor(R.color.primary_dark));

    mgr=(MediaProjectionManager)getSystemService(MEDIA_PROJECTION_SERVICE);
  }
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/MainActivity.java)

`MediaProjectionManager`, at the time of this writing (October 2015), has a grand total of *two* methods. When the user taps on the device action bar item, we invoke fully 50% of the `MediaProjectionManager`, calling `createScreenCaptureIntent()`. This will return an `Intent`, designed to be used with `startActivityForResult()`, that brings up the screenshot permission dialog:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId()==R.id.start) {
      startActivityForResult(mgr.createScreenCaptureIntent(),
          REQUEST_SCREENSHOT);
    }
    else {
      stopService(new Intent(this, ProjectorService.class));
    }

    return super.onOptionsItemSelected(item);
  }
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/MainActivity.java)

In `onActivityResult()`, if our request for permission was granted, we pass the details along via `Intent` extras to a `ProjectorService` that we start using `startService()`:

```
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==REQUEST_SCREENSHOT) {
      if (resultCode==RESULT_OK) {
        Intent i=
            new Intent(this, ProjectorService.class)
                .putExtra(ProjectorService.EXTRA_RESULT_CODE,
                  resultCode)
                .putExtra(ProjectorService.EXTRA_RESULT_INTENT,
                  data);

        startService(i);
```

```
    }
  }
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/MainActivity.java)

The rest of the `MainActivity` is mostly doing the same sort of work as was seen in the sample apps from the chapter on embedding a Web server, including populating the `ListView` with the URLs for our projection.

Creating the MediaProjection

`ProjectorService` extends `WebServerService`, our reusable embedded Web server. However, most of its business logic — along with code extracted into a separate `ImageTransmogrifier` — involves fetching screenshots using the media projection APIs, generating PNGs for them, and pushing them over to the Web browser.

In `onCreate()` of `ProjectorService`, we:

- get our hands on a `MediaProjectionManager` and a `WindowManager` system service
- set up a `HandlerThread` and create an associated `Handler` for it, as the media projection process wants a `Handler`

```
@Override
public void onCreate() {
  super.onCreate();

  mgr=(MediaProjectionManager)getSystemService(MEDIA_PROJECTION_SERVICE);
  wmgr=(WindowManager)getSystemService(WINDOW_SERVICE);

  handlerThread.start();
  handler=new Handler(handlerThread.getLooper());
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

That `HandlerThread` is created in an initializer, since it does not directly depend on a `Context`:

```
final private HandlerThread handlerThread=new HandlerThread(getClass().getSimpleName(),
    android.os.Process.THREAD_PRIORITY_BACKGROUND);
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

In `onStartCommand()`, we then use the remaining 50% of the `MediaProjectionService` API to get a `MediaProjection`, using the values that were passed to `onActivityResult()` from our permission request which, in turn, were passed to `ProjectorService` via `Intent` extras:

```
    projection=
        mgr.getMediaProjection(i.getIntExtra(EXTRA_RESULT_CODE, -1),
            (Intent)i.getParcelableExtra(EXTRA_RESULT_INTENT));
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

We then create an instance of `ImageTransmogrifier`, passing in the `ProjectorService` itself as a constructor parameter:

```
    it=new ImageTransmogrifier(this);
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

`ImageTransmogrifier`, in its constructor, sets about determining the screen size (using `WindowManager` and `getDefaultDisplay()`). Since high-resolution displays will wind up with *very* large bitmaps, and therefore slow down the data transfer, we scale the width and height until such time as each screenshot will contain no more than 512K pixels.

```
public class ImageTransmogrifier implements ImageReader.OnImageAvailableListener {
  private final int width;
  private final int height;
  private final ImageReader imageReader;
  private final ProjectorService svc;
  private Bitmap latestBitmap=null;

  ImageTransmogrifier(ProjectorService svc) {
    this.svc=svc;

    Display display=svc.getWindowManager().getDefaultDisplay();
    Point size=new Point();

    display.getSize(size);

    int width=size.x;
    int height=size.y;

    while (width*height > (2<<19)) {
      width=width>>1;
      height=height>>1;
    }

    this.width=width;
    this.height=height;

    imageReader=ImageReader.newInstance(width, height,
        PixelFormat.RGBA_8888, 2);
```

```
    imageReader.setOnImageAvailableListener(this, svc.getHandler());
  }
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ImageTransmogrifier.java)

Finally, we create a new `ImageReader`, which boils down to a class that manages a bitmap `Surface` that can be written to, using our specified width, height, and bit depth. In particular, we are saying that there are two possible outstanding bitmaps at a time, courtesy of the `2` final parameter, and that we should be notified when a new image is ready, by registering the `ImageTransmogrifier` as the listener. The `Handler` is used so that we are informed about image availability on our designated background `HandlerThread`.

Back over in `ProjectorService`, we then as the `MediaProjection` to create a `VirtualDisplay`, tied to the `ImageTransmogrifier` and its `ImageReader`:

```
    vdisplay=projection.createVirtualDisplay("andprojector",
        it.getWidth(), it.getHeight(),
        getResources().getDisplayMetrics().densityDpi,
        VIRT_DISPLAY_FLAGS, it.getSurface(), null, handler);
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

We need to provide:

- a name for this virtual display, primarily for logging purposes
- the size of the virtual display, in terms of width and height, where we use the scaled width and height computed by the `ImageTransmogrifier`
- the density of the virtual display, which we set to match the density of the actual device screen
- a set of flags (`VIRT_DISPLAY_FLAGS`), where the magic values that seem to work are `VIRTUAL_DISPLAY_FLAG_OWN_CONTENT_ONLY` and `VIRTUAL_DISPLAY_FLAG_PUBLIC`:

```
static final int VIRT_DISPLAY_FLAGS=
    DisplayManager.VIRTUAL_DISPLAY_FLAG_OWN_CONTENT_ONLY |
    DisplayManager.VIRTUAL_DISPLAY_FLAG_PUBLIC;
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

- a `Surface` representing the virtual display, in this case retrieved from the `ImageReader` inside the `ImageTransmogrifier`

```
Surface getSurface() {
  return(imageReader.getSurface());
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ImageTransmogrifier.java)

- an optional `VirtualDisplay.Callback` to be notified about events in the lifecycle of the `VirtualDisplay` (unused here, so we pass `null`)
- a `Handler` from a `HandlerThread`, to be used for that callback (presumably unused here, but since we have the right `Handler` anyway, we use it)

We also need to know about events surrounding the `MediaProjection` itself, so we create and register a `MediaProjection.Callback`, as part of the full `onStartCommand()` implementation:

```
@Override
public int onStartCommand(Intent i, int flags, int startId) {
  projection=
      mgr.getMediaProjection(i.getIntExtra(EXTRA_RESULT_CODE, -1),
          (Intent)i.getParcelableExtra(EXTRA_RESULT_INTENT));

  it=new ImageTransmogrifier(this);

  MediaProjection.Callback cb=new MediaProjection.Callback() {
    @Override
    public void onStop() {
      vdisplay.release();
    }
  };

  vdisplay=projection.createVirtualDisplay("andprojector",
      it.getWidth(), it.getHeight(),
      getResources().getDisplayMetrics().densityDpi,
      VIRT_DISPLAY_FLAGS, it.getSurface(), null, handler);
  projection.registerCallback(cb, handler);

  return(START_NOT_STICKY);
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

And, at this point, the device will start collecting screenshots for us.

Processing the Screenshots

Of course, it would be useful if we could actually receive those screenshots and do something with them.

We find out when a screenshot is available via the `ImageReader.Callback` we set up in `ImageTransmogrifier`, specifically its `onImageAvailable()` callback. Since `ImageTransmogrifier` itself is implementing the `ImageReader.Callback` interface, `ImageTransmogrifier` has the `onImageAvailable()` implementation:

```
@Override
public void onImageAvailable(ImageReader reader) {
  final Image image=imageReader.acquireLatestImage();

  if (image!=null) {
    Image.Plane[] planes=image.getPlanes();
    ByteBuffer buffer=planes[0].getBuffer();
    int pixelStride=planes[0].getPixelStride();
    int rowStride=planes[0].getRowStride();
    int rowPadding=rowStride - pixelStride * width;
    int bitmapWidth=width + rowPadding / pixelStride;

    if (latestBitmap == null ||
        latestBitmap.getWidth() != bitmapWidth ||
        latestBitmap.getHeight() != height) {
      if (latestBitmap != null) {
        latestBitmap.recycle();
      }

      latestBitmap=Bitmap.createBitmap(bitmapWidth,
          height, Bitmap.Config.ARGB_8888);
    }

    latestBitmap.copyPixelsFromBuffer(buffer);

    if (image != null) {
      image.close();
    }

    ByteArrayOutputStream baos=new ByteArrayOutputStream();
    Bitmap cropped=Bitmap.createBitmap(latestBitmap, 0, 0,
      width, height);

    cropped.compress(Bitmap.CompressFormat.PNG, 100, baos);

    byte[] newPng=baos.toByteArray();
```

```
      svc.updateImage(newPng);
    }
  }
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ImageTransmogrifier.java)

This is complex.

First, we ask the `ImageReader` for the latest image, via `acquireLatestImage()`. If, for some reason, there is no image, there is nothing for us to do, so we skip all the work.

Otherwise, we have to go through some gyrations to get the actual bitmap itself from `Image` object. The recipe for that probably makes sense to somebody, but that "somebody" is not the author of this book. Suffice it to say, the first six lines of the main `if` block in `onImageAvaialble()` get access to the bytes of the bitmap (as a `ByteBuffer` named `buffer`) and determine the width of the bitmap that was handed to us (as an `int` named `bitmapWidth`).

Because `Bitmap` objects are large and therefore troublesome to allocate, we try to reuse one where possible. If we do not have a `Bitmap` (`latestBitmap`), or if the one we have is not the right size, we create a new `Bitmap` of the appropriate size. Otherwise, we use the `Bitmap` that we already have. Regardless of where the `Bitmap` came from, we use `copyPixelsFromBuffer()` to populate it from the `ByteBuffer` we got from the `Image.Plane` that we got from the `Image` that we got from the `ImageReader`.

You might think that this `Bitmap` would be the proper size. However, it is not. For inexplicable reasons, it will be a bit larger, with excess unused pixels on each row on the end. This is why we need to use `Bitmap.createBitmap()` to create a cropped edition of the original `Bitmap`, for our actual desired width.

We then `compress()` the cropped `Bitmap` into a PNG file, get the `byte` array of pixel data from the compressed result, and hand that off to the `ProjectorService` via `updateImage()`.

`updateImage()`, in turn, holds onto this most-recent PNG file in an `AtomicReference` wrapped around the `byte` array:

```
  private AtomicReference<byte[]> latestPng=new AtomicReference<byte[]>();
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

This way, when some Web server thread goes to serve up this PNG file, we do not have to worry about thread contention with the `HandlerThread` we are using for the screenshots themselves.

Then, we iterate over all connected browsers' WebSocket connections and send a unique URL to them, where the uniqueness (from `SystemClock.uptimeMillis()`) is designed as a "cache-busting" approach to ensure the browser always requests the image

```
void updateImage(byte[] newPng) {
  latestPng.set(newPng);

  for (WebSocket socket : getWebSockets()) {
    socket.send("screen/"+Long.toString(SystemClock.uptimeMillis()));
  }
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

Those WebSockets are enabled by `ProjectorService` calling `serveWebSockets()` on its `WebServerService` superclass, in the `configureRoutes()` callback:

```
@Override
protected boolean configureRoutes(AsyncHttpServer server) {
  serveWebSockets("/ss", null);

  server.get(getRootPath()+"/screen/.*",
    new ScreenshotRequestCallback());

  return(true);
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

The `ScreenshotRequestCallback` is an inner class of `ProjectorService`, one that serves the PNG file itself in response to a request:

```
private class ScreenshotRequestCallback
    implements HttpServerRequestCallback {
  @Override
  public void onRequest(AsyncHttpServerRequest request,
                        AsyncHttpServerResponse response) {
    response.setContentType("image/png");

    byte[] png=latestPng.get();
    ByteArrayInputStream bais=new ByteArrayInputStream(png);
```

```
      response.sendStream(bais, png.length);
    }
  }
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

The result is that, whenever a screenshot is ready, we create the PNG file and tell the browser "hey! we have an update!".

The HTML

The Web content that is served to the browser is reminiscent of the HTML and JavaScript used in the section on implementing WebSockets. There, the messages being pushed to the browser were timestamps, shown in a list. Here, the messages being pushed to the browser are URLs to load a fresh screenshot.

Hence, the HTML just has an `<img>` tag for that screenshot, with an `id` of `screen`, loading `screen/0` at the outset to bootstrap the display:

```
<html>
<head>
    <title>andprojector</title>
</head>
<body>
<img id="screen"
  style="height: 100%; width: 100%; object-fit: contain"
  src="screen/0">
<script src="js/app.js"></script>
</body>
</html>
```

(from MediaProjection/andprojector/app/src/main/assets/index.html)

The JavaScript registers for a WebSocket connection, then updates that `<img>` with a fresh URL when such a URL is pushed over to the browser:

```
window.onload = function() {
    var screen=document.getElementById('screen');
    var ws_url=location.href.replace('http://', 'ws://')+'ss';
    var socket=new WebSocket(ws_url);

    socket.onopen = function(event) {
      // console.log(event.currentTarget.url);
    };
```

```
    socket.onerror = function(error) {
      console.log('WebSocket error: ' + error);
    };

    socket.onmessage = function(event) {
      screen.src=event.data;
    };
}
```

(from MediaProjection/andprojector/app/src/main/assets/js/app.js)

Of course, in principle, there could be much more to the Web UI, including some ability to stop all of this when it is no longer needed. Speaking of which...

Shutting Down

The user can stop the screenshot collection and broadcasting either via the action bar item or the action in the `Notification` that is raised in support of the foreground service. In either case, in `onDestroy()`, in addition to chaining to `WebServerService` to shut down the Web server, `ProjectorService` stops the `MediaProjection`:

```
  @Override
  public void onDestroy() {
    projection.stop();

    super.onDestroy();
  }
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

This should also trigger our `VirtualDisplay.Callback`, causing us to release the `VirtualDisplay`.

Dealing with Configuration Changes

However, there is one interesting wrinkle we have to take into account: what happens if the user rotates the screen? We need to update our `VirtualDisplay` and `ImageReader` to take into account the new screen height and width.

`ProjectorService` will be called with `onConfigurationChanged()` when any configuration change occurs. This could be due to a screen rotation or other triggers (e.g., putting the device into a car dock). So, we need to see if the screen height or

width changed — if not, we do not need to do anything. So, we create a new `ImageTransmogrifier` and compare its height and width to the current height and width:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);

  ImageTransmogrifier newIt=new ImageTransmogrifier(this);

  if (newIt.getWidth()!=it.getWidth() ||
    newIt.getHeight()!=it.getHeight()) {
    ImageTransmogrifier oldIt=it;

    it=newIt;
    vdisplay.resize(it.getWidth(), it.getHeight(),
      getResources().getDisplayMetrics().densityDpi);
    vdisplay.setSurface(it.getSurface());

    oldIt.close();
  }
}
```

(from MediaProjection/andprojector/app/src/main/java/com/commonsware/andprojector/ProjectorService.java)

If a dimension has changed, we tell the `VirtualDisplay` to resize to the new height and width, attach a new `Surface` from the new `ImageReader`, and switch over to the new `ImageTransmogrifier`, closing the old one.

This solution is not perfect — there is a bit of a race condition if a screenshot is taken while the configuration change is going on – but for a non-production-grade app it will suffice.

Recording the Screen

Here, a "screencast" refers to a full-motion video of what goes on the screen. You can think of it as a series of screenshots all written to one video file (e.g., an MP4). Many apps on the Play Store have screencasts as part of their product profile, so you can see what the app looks like when it is run.

Android's media projection APIs allow you to capture screencasts, using a mechanism similar to the one used to take screenshots. You have to ask permission from the user to be able to record the screen, and that permission will last for the

duration of one screen recording. During that period of time, you can direct Android to make a duplicate copy of what goes on the screen to a video file. This winds up using the `MediaRecorder` API along with dedicated media projection APIs, which is a bit awkward, since `MediaRecorder` is really aimed at using the device camera to record videos of the world outside the device.

Jake Wharton, with his open source Telecine app, helped blaze the trail in how these APIs are supposed to work, since the documentation, as usual, is limited.

This chapter will examine a separate app, `MediaProjection/andcorder`, that offers screen recording through the media projection APIs. In the end, andcorder does the same basic stuff as does Telecine, with fewer bells and whistles. Also, the control channel is different: Telecine uses a screen overlay, while andcorder uses a foreground `Notification` or the command line.

Requesting Media Projection… Without a GUI

The andprojector sample app profiled earlier in this chapter used the media projection APIs, just as andcorder does. Both have to do the same work at the outset: ask the user for permission to record the screen. In the case of andprojector, while we had a foreground `Notification` to stop the projection, starting the projection was done through the andprojector activity, via an action bar item. The andcorder app, on the other hand, will demonstrate a different approach to this… and highlight a regression introduced in Android 6.0.

`MainActivity` is designed to be an invisible activity, like a few others used elsewhere in this book. We want a launcher icon in the home screen to be able to initialize the app, but we do not need an activity's UI to control it.

So, we skip the `setContentView()` call, and in `onCreate()` just `call`startActivityForResult(), `using` theIntentsupplied by createScreenCaptureIntent()`from` aMediaProjectionManager`:

```
package com.commonsware.android.andcorder;

import android.app.Activity;
import android.content.Intent;
import android.media.projection.MediaProjectionManager;
import android.os.Bundle;

public class MainActivity extends Activity {
  private static final int REQUEST_SCREENCAST=59706;
  private MediaProjectionManager mgr;

  @Override
```

```
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mgr=(MediaProjectionManager)getSystemService(MEDIA_PROJECTION_SERVICE);

    startActivityForResult(mgr.createScreenCaptureIntent(),
      REQUEST_SCREENCAST);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode==REQUEST_SCREENCAST) {
      if (resultCode==RESULT_OK) {
        Intent i=
            new Intent(this, RecorderService.class)
                .putExtra(RecorderService.EXTRA_RESULT_CODE, resultCode)
                .putExtra(RecorderService.EXTRA_RESULT_INTENT, data);

        startService(i);
      }
    }

    finish();
  }
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/MainActivity.java)

In `onActivityResult()`, we just pass the data along to a `RecorderService`, which is responsible for starting and stopping the screen recording. Then, we `finish()` the activity, as it is no longer needed.

This looks simple enough. It even works well, on Android 5.0 and 5.1. On Android 6.0, though, we have some problems.

The activity is designed to be used with `Theme.Translucent.NoTitleBar`, as the other "invisible activity" book samples use. Most of those samples will work just fine on Android 6.0. In particular, a `Theme.Translucent.NoTitleBar` activity that does its work in `onCreate()` and then calls `finish()` should be just fine on Android 6.0.

But sometimes the work that needs to be done is a bit more involved than that. In particular, calling `startActivityForResult()`, with an eye towards calling `finish()` in `onActivityResult()`, will cause your app to crash with an `IllegalStateException` saying that your activity "did not call finish() prior to onResume() completing". This, apparently, is a requirement of `Theme.Translucent.NoTitleBar` activities on Android 6.0+.

So, we have to things a bit differently, to accommodate this undocumented regression in behavior.

Rather than refer to `Theme.Translucent.NoTitleBar` directly in the manifest, we refer to a custom `Theme.Apptheme` resource instead:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  package="com.commonsware.android.andcorder"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".MainActivity"
      android:theme="@style/AppTheme">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <service
      android:name=".RecorderService"
      android:exported="true"/>

  </application>

</manifest>
```

(from MediaProjection/andcorder/app/src/main/AndroidManifest.xml)

This custom theme inherits from `Theme.Translucent.NoTitleBar`, in `res/values/styles.xml`:

```
<resources>

  <style name="AppTheme" parent="android:Theme.Translucent.NoTitleBar">
  </style>

</resources>
```

(from MediaProjection/andcorder/app/src/main/res/values/styles.xml)

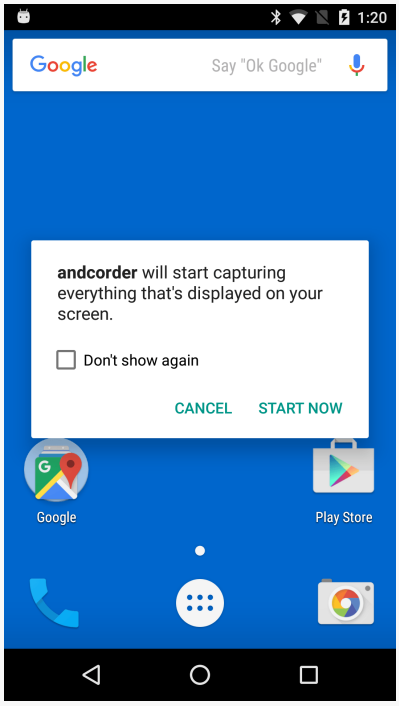


Figure 856: andcorder, As Initially Launched

Implementing a Control Channel… Without a GUI

We need to be able to tell andcorder to start and stop screen recording. If we are going to have an invisible activity, we need some other way to tell andcorder what it is supposed to do.

One approach used in andcorder is a `Notification`, tied to the foreground service that manages the actual screen recording.

We will use action strings, in the `Intent` used to start the `RecorderService`, to indicate what is to be done. Those action strings will be the application ID plus a segment at the end that is the specific operation we want:

```
static final String ACTION_RECORD=
  BuildConfig.APPLICATION_ID+".RECORD";
static final String ACTION_STOP=
  BuildConfig.APPLICATION_ID+".STOP";
static final String ACTION_SHUTDOWN=
  BuildConfig.APPLICATION_ID+".SHUTDOWN";
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

Here, we use `BuildConfig.APPLICATION_ID`, a faster, no-`Context` way to get our application ID, as part of building up these strings. We have three actions: to start recording (`RECORD`), to stop recording (`STOP`), and to shut down the `RecorderService` (`SHUTDOWN`). An `Intent` with *no* action string will be used on the initial launch of the service, from `MainActivity`.

`onStartCommand()` is where all of these commands, triggered by `startService()` calls, will come in:

```
@Override
public int onStartCommand(Intent i, int flags, int startId) {
  if (i.getAction()==null) {
    resultCode=i.getIntExtra(EXTRA_RESULT_CODE, 1337);
    resultData=i.getParcelableExtra(EXTRA_RESULT_INTENT);

    if (recordOnNextStart) {
      startRecorder();
    }

    foregroundify(!recordOnNextStart);
    recordOnNextStart=false;
  }
  else if (ACTION_RECORD.equals(i.getAction())) {
    if (resultData!=null) {
      foregroundify(false);
      startRecorder();
    }
    else {
      Intent ui=
        new Intent(this, MainActivity.class)
          .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

      startActivity(ui);
      recordOnNextStart=true;
    }
  }
  else if (ACTION_STOP.equals(i.getAction())) {
    foregroundify(true);
    stopRecorder();
  }
  else if (ACTION_SHUTDOWN.equals(i.getAction())) {
    stopSelf();
  }

  return(START_NOT_STICKY);
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

If we have no action string, this should be the command from `MainActivity`, so we grab the `resultCode` and `resultData` out of the `Intent` and stash them in simple fields on the service:

```
  private int resultCode;
  private Intent resultData;
  private boolean recordOnNextStart=false;
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

We also:

- Call `startRecorder()` if `recordOnNextStart` is set to `true`
- Call `foregroundify()`, with a `boolean` that indicates whether we should give the user the option to begin recording (`true`) or to stop existing recording (`false`)
- Clear the `recordOnNextStart` flag

We will discuss more about that `recordOnNextStart`, its role, and why it exists, later in this chapter.

If, instead, a `RECORD` action string was on the `Intent`, then ideally we should begin recording the screen contents. The "ideally" part is because there will be scenarios in which the `RECORD` action is invoked before we actually have permission from the user to record the screen (more on this later).

So, if a `RECORD` action comes in, *and* we have permission from the user to record the screen (`resultData` is not `null`), we call `startRecorder()` to start recording, plus call `foregroundify()` to put up a `Notification` with an action for `STOP`. If, on the other hand, we do not presently have permission from the user (`resultData` is `null`), we start up `MainActivity` to get that permission, plus set `recordOnNextStart` to `true`.

The other two cases are simpler:

- If we get a `STOP` `Intent`, we call `stopRecorder()`, plus call `foregroundify()` to change the foreground service `Notification` to one that has an action for `RECORD`
- If we get a `SHUTDOWN` `Intent`, we call `stopSelf()` to go away entirely

`foregroundify()` is invoked for most of those cases, to put the service in the foreground (if it is not in the foreground already) and show a `Notification` with the appropriate mix of actions:

```
private void foregroundify(boolean showRecord) {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this);

  b.setAutoCancel(true)
    .setDefaults(Notification.DEFAULT_ALL);

  b.setContentTitle(getString(R.string.app_name))
    .setSmallIcon(R.mipmap.ic_launcher)
    .setTicker(getString(R.string.app_name));

  if (showRecord) {
    b.addAction(R.drawable.ic_videocam_white_24dp,
      getString(R.string.notify_record), buildPendingIntent(ACTION_RECORD));
  }
  else {
    b.addAction(R.drawable.ic_stop_white_24dp,
      getString(R.string.notify_stop), buildPendingIntent(ACTION_STOP));
  }

  b.addAction(R.drawable.ic_eject_white_24dp,
    getString(R.string.notify_shutdown), buildPendingIntent(ACTION_SHUTDOWN));

  if (isForeground) {
    NotificationManager mgr=(NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    mgr.notify(NOTIFY_ID, b.build());
  }
  else {
    startForeground(NOTIFY_ID, b.build());
    isForeground=true;
  }
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

In addition to generic `NotificationCompat.Builder` configuration, we:

- add an action to shut down the service, tied to the `SHUTDOWN` action string
- either add an action to `RECORD` or `STOP` the recording, based upon the `boolean` passed into `foregroundify()`
- either use `startForeground()` to move the service into the foreground and show the `Notification` or use `NotificationManager` to update the existing `Notification` (if we are already in the foreground)

The latter distinction may not be necessary. Calling `startForeground()` multiple times does not seem to have any harm, and it also updates the foreground

`Notification`. Using `NotificationManager` directly for the already-in-the-foreground scenario, though, may be superfluous.

The `addAction()` calls delegate to a `buildPendingIntent()` method, to create the `PendingIntent` to be triggered when the action is tapped:

```
private PendingIntent buildPendingIntent(String action) {
  Intent i=new Intent(this, getClass());

  i.setAction(action);

  return(PendingIntent.getService(this, 0, i, 0));
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

This creates an explicit `Intent`, tied to `RecorderService` itself, but also adds the action string. This `Intent` will always resolve to our `RecorderService`; the action string is just part of the payload.

That foreground `Notification` provides the visual way of starting recording:

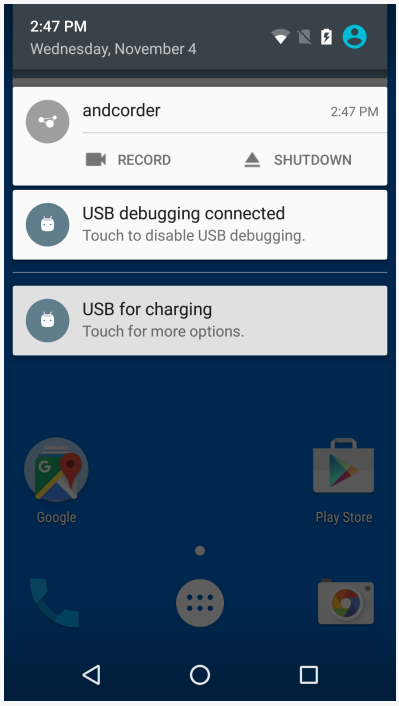


Figure 857: andcorder Notification, Showing Record and Shutdown Actions

...and stopping recording once started:

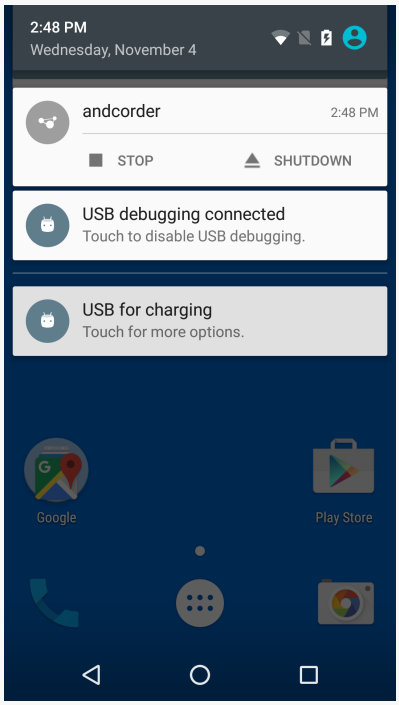


Figure 858: andcorder Notification, Showing Stop and Shutdown Actions

In addition, `onDestroy()` stops the recording and removes us from the foreground, plus we have the obligatory `onBind()` implementation:

```
@Override
public void onDestroy() {
  stopRecorder();
  stopForeground(true);

  super.onDestroy();
}

@Override
public IBinder onBind(Intent intent) {
  throw new IllegalStateException("go away");
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

Using the Control Channel… From the Command Line

The downside of relying upon a foreground `Notification` is that the user has to interact with that `Notification` to start and stop the recording. As a result, that `Notification` — and the rest of the notification tray — will be visible at the beginning and the end of the recording. While this could be addressed by editing the video, video editors can be difficult to use. It would be nice to be able to operate andcorder without affecting the screen.

Fortunately, we can, courtesy of `adb`.

As is covered in the chapter on ADB, it is possible to use the `adb shell am` command to start an activity, start a service, and send a broadcast. In this case, since we are using a service for managing the recording process, we can use `adb shell am` to trigger the same actions that the `Notification` does.

This, however, requires that our `RecorderService` be exported. For the `PendingIntent` objects used in the `Notification`, we would not need to export the service. Invoking the service from the command line, however, does require an exported service, since the command line is not the app itself and therefore is considered to be a third-party client of the app. Moreover, there is no obvious way to validate that the commands were sent from `adb shell am`, which means that when andcorder is installed, any app could send commands to `RecorderService`.

From a security standpoint, this is not great. The user still has to be involved to grant permission to record the screen, which limits the security risk a little bit. However, in general, **you should not run andcorder on your own personal device**, due to this security hole. Or, at minimum, run andcorder, then uninstall it immediately when you are done with it, so it does not linger where malware might try to use it.

The andcorder project contains three `bash` scripts to invoke the `RecorderService`. These should be able to be trivially converted to Windows command files; the proof of this is left as an exercise for the reader.

All three scripts use `adb shell am startservice`, and all point to the same component (`-n com.commonsware.android.andcorder/.RecorderService`). What varies is the action string supplied to the `-a` switch.

NOTE: the shell script code listings are word-wrapped due to line length limitations in the books; the files themselves have the `adb shell` commands all on one line.

So, the `record` script, for example, passes `com.commonsware.android.andcorder.RECORD` as the action string:

```
#!/bin/bash

adb shell am startservice -n com.commonsware.android.andcorder/.RecorderService
-a com.commonsware.android.andcorder.RECORD
```

The `stop` script passes the `STOP` action string; the `shutdown` script passes the `SHUTDOWN` action string.

These, therefore, replicate the `Intent` structures used in the `PendingIntent` objects for the `Notification` actions.

However, there is one key usage difference: it would be nice to be able to run the `record` script without having to think about whether or not you ran andcorder from the home screen launcher or not. The `RECORD` action cannot actually do the recording without the result data from the `startActivityForResult()` call in `MainActivity`.

This is why the `RECORD` action logic detects this case and starts up `MainActivity` — so we can just run the `record` script and, if we do not presently have screen-recording permission, request it from the user.

The `recordOnNextStart` flag indicates whether or not `RECORD` started up `MainActivity`. If it did, when we get the result data in the no-action `onStartCommand()` call, we should go ahead and begin recording. This prevents the user from having to run the `record` script twice, once to pop up the permission dialog and once to actually begin recording.

Starting the Recording

The `startRecorder()` method on `RecorderService` is called when it is time to begin screen recording, either because the user asked us to record just now or the user asked us to record (via the command-line script) and we just now got permission from the user to do that.

```
  synchronized private void startRecorder() {
    if (session==null) {
      MediaProjectionManager mgr=
        (MediaProjectionManager)getSystemService(MEDIA_PROJECTION_SERVICE);
      MediaProjection projection=
```

```
      mgr.getMediaProjection(resultCode, resultData);

    session=
      new RecordingSession(this, new RecordingConfig(this),
        projection);
    session.start();
  }
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

Here, as with the andprojector sample, we use a `MediaProjectionManager` to turn the `resultCode` `int` and `resultData` `Intent` into a `MediaProjection`. Then, we create a `RecordingSession`, wrapped around a `RecordingConfig` and the `MediaProjection`, and call `start()` on the `RecordingSession`.

Both `RecordingSession` and `RecordingConfig` are classes that are part of the app, not the Android SDK. `RecordingConfig` holds onto information about the nature of what is being recorded (notably, the video resolution) to capture. `RecordingSession` handles the stateful work of actually recording the video.

Of the two, you might expect `RecordingSession` to be far more complex. In truth, it is decidedly more straightforward than is `RecordingConfig`. Determining the resolution and other information about our screen recording is annoyingly complicated.

Deciding How Big Our Recording Is

The job of `RecordingConfig` is to derive and hold onto five pieces of data regarding the screen recording that we are about to initiate:

- The width and height of the video, in pixels
- The bit rate at which the video should be recorded
- The frame rate (frames per second) at which the video should be recorded
- The screen density

These are held in five `final int` fields, as `RecordingConfig` is designed to be immutable:

```
final int width;
final int height;
final int frameRate;
```

```
  final int bitRate;
  final int density;
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingConfig.java)

All five of these values will be initialized in the constructor (since they are `final`). In fact, all the business logic for `RecordingSession` is just in the constructor, to derive these five values.

That constructor starts off simple enough:

```
RecordingConfig(Context ctxt) {
  DisplayMetrics metrics=new DisplayMetrics();
  WindowManager wm=(WindowManager)ctxt.getSystemService(Context.WINDOW_SERVICE);

  wm.getDefaultDisplay().getRealMetrics(metrics);

  density=metrics.densityDpi;

  Configuration cfg=ctxt.getResources().getConfiguration();

  boolean isLandscape=
    (cfg.orientation==Configuration.ORIENTATION_LANDSCAPE);
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingConfig.java)

Here, we:

- Populate a `DisplayMetrics` data structure, given a `WindowManager`
- Save the screen density in its `final` field
- Get the current `Configuration` and determine if we are in landscape mode or not

Where things start to get messy is with the other four fields, as they need to be populated based on the device's video recording capabilities. For various reasons, screen recording is actually handled mostly by `MediaRecorder`, the same class used to record videos from a device camera. Hence, we are limited by not only the actual resolution of the screen but by the capabilities of the video recording engine.

The classic way to handle this is by using `CamcorderProfile` objects. These standardize video recording support for various resolutions. We can find out which of these profiles the device supports and use that to help determine our video resolution, frame rate, and bitrate.

However, we also have to take into account the resolution of the screen itself. If `MediaRecorder` is capable of 1080p (1920 x 1080) video recording, but the device has a low-end WXGA (1280 x 800) screen, we will waste a lot of space recording that

screen at 1080p. What we want is the smallest resolution that is bigger than the screen, to minimize wasted space while not losing data. If, for some reason, we do not have a `CamcorderProfile` that is bigger than the screen, we will have to settle for one that is as big as we can manage.

To that end, the `CAMCORDER_PROFILES` static field on `RecordingConfig` lists the major `CamcorderProfile` IDs, in descending order based on resolution:

```
private static final int[] CAMCORDER_PROFILES={
  CamcorderProfile.QUALITY_2160P,
  CamcorderProfile.QUALITY_1080P,
  CamcorderProfile.QUALITY_720P,
  CamcorderProfile.QUALITY_480P,
  CamcorderProfile.QUALITY_CIF,
  CamcorderProfile.QUALITY_QVGA,
  CamcorderProfile.QUALITY_QCIF
};
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingConfig.java)

If we simply iterate over this list and choose either the first one we find, or one that is smaller yet is bigger than the screen, we will get the right `CamcorderProfile` for our use case:

```
  CamcorderProfile selectedProfile=null;

  for (int profileId : CAMCORDER_PROFILES) {
    CamcorderProfile profile=null;

    try {
      profile=CamcorderProfile.get(profileId);
    }
    catch (Exception e) {
      // not documented to throw anything, but does
    }

    if (profile!=null) {
      if (selectedProfile==null) {
        selectedProfile=profile;
      }
      else if (profile.videoFrameWidth>=metrics.widthPixels &&
        profile.videoFrameHeight>=metrics.heightPixels) {
        selectedProfile=profile;
      }
    }
  }
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingConfig.java)

To get a `CamcorderProfile` given its ID, you call the static `get()` method on `CamcorderProfile`. This is supposed to return the `CamcorderProfile` if it is supported or `null` if it is not. In actuality, it may throw an exception if the profile is not supported, which is why we have to wrap the `get()` call in a `try/catch` block. Then, if profile exists, we hold onto it as the `selectedProfile` if either:

- `selectedProfile` is `null`, meaning this is the largest available profile, or
- the profile has a resolution bigger than the screen on both axes

If, after all that is done, we have a `null` `selectedProfile`, that means that *none* of the `CamcorderProfile` values were available. That is very strange, and rather than take a random guess as to what will work, we just blow up with an `IllegalStateException`. Obviously, a production-grade app would need to blow up more nicely.

Otherwise, we can collect our remaining data... which once again is more complex than you might expect:

```
    if (selectedProfile==null) {
      throw new IllegalStateException("No CamcorderProfile available!");
    }
    else {
      frameRate=selectedProfile.videoFrameRate;
      bitRate=selectedProfile.videoBitRate;

      int targetWidth, targetHeight;

      if (isLandscape) {
        targetWidth=selectedProfile.videoFrameWidth;
        targetHeight=selectedProfile.videoFrameHeight;
      }
      else {
        targetWidth=selectedProfile.videoFrameHeight;
        targetHeight=selectedProfile.videoFrameWidth;
      }

      if (targetWidth>=metrics.widthPixels &&
        targetHeight>=metrics.heightPixels) {
        width=metrics.widthPixels;
        height=metrics.heightPixels;
      }
      else {
        if (isLandscape) {
```

```
        width=targetHeight*metrics.widthPixels/metrics.heightPixels;
        height=targetHeight;
      }
      else {
        width=targetWidth;
        height=targetWidth*metrics.heightPixels/metrics.widthPixels;
      }
    }
  }
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingConfig.java)

Getting the frame rate and the bitrate are easy enough, as they are just fields on the `CamcorderProfile`. Where things start to get strange is in determining what we should tell the `MediaRecorder` that we want recorded in terms of resolution.

Partly, this is a problem of orientation. `MediaRecorder` thinks that everything is recorded in landscape, but we may well want to record the screen held in portrait mode.

Partly, this is a problem of aspect ratios. There is no requirement that the `MediaRecorder` advertise support for resolutions that match the screen size, or even match the screen's aspect ratio. So, if the `MediaRecorder` is capable of recording our full screen, we ask it to record the full screen (as determined from the `DisplayMetrics`). If, however, we are on some odd device whose `MediaRecorder` is not capable of recording video at the screen's own resolution, we try to at least maintain the aspect ratio of the screen when deriving the resolution to use for recording.

The net of all that work is that we have the details of how we want the screen recording to be done, encapsulated in the `RecordingConfig` object, ready for use by the `RecordingSession`.

Actually Recording Stuff

None of that actually records the screen, though. That is the responsibility of the `RecordingSession`.

In the `RecordingSession` constructor, we:

- Hold onto the `RecordingConfig` and `MediaProjection`
- Hold onto the application `Context`, as we will need a `Context` later on

- Create an instance of a `ToneGenerator` to use for audible feedback about the state of the recording
- Create a `File` object pointing at our desired output: an `andcorder.mp4` file in our app's portion of external storage

```
RecordingSession(Context ctxt, RecordingConfig config,
                 MediaProjection projection) {
  this.ctxt=ctxt.getApplicationContext();
  this.config=config;
  this.projection=projection;
  this.beeper=new ToneGenerator(
    AudioManager.STREAM_NOTIFICATION, 100);

  output=new File(ctxt.getExternalFilesDir(null), "andcorder.mp4");
  output.getParentFile().mkdirs();
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingSession.java)

The actual work to record the video is handled in the `start()` method on `RecordingSession`, where we set up the `MediaRecorder` and a `VirtualDisplay`, the latter being the same thing that we used in the andprojector sample:

```
void start() {
  recorder=new MediaRecorder();
  recorder.setVideoSource(MediaRecorder.VideoSource.SURFACE);
  recorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
  recorder.setVideoFrameRate(config.frameRate);
  recorder.setVideoEncoder(MediaRecorder.VideoEncoder.H264);
  recorder.setVideoSize(config.width, config.height);
  recorder.setVideoEncodingBitRate(config.bitRate);
  recorder.setOutputFile(output.getAbsolutePath());

  try {
    recorder.prepare();
    vdisplay=projection.createVirtualDisplay("andcorder",
      config.width, config.height, config.density,
      VIRT_DISPLAY_FLAGS, recorder.getSurface(), null, null);
    beeper.startTone(ToneGenerator.TONE_PROP_ACK);
    recorder.start();
  }
  catch (IOException e) {
    throw new RuntimeException("Exception preparing recorder", e);
  }
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingSession.java)

First, we create an instance of `MediaRecorder` and configure it. As is discussed in the chapter on working with the camera, `MediaRecorder` is a *very* fussy class, requiring a fairly specific order of method calls to configure it without messing things up too bad. The values for the configuration come from:

- the `RecordingConfig`, notably the requested resolution, frame rate, and bitrate
- the output `File` created in the `RecordingSession` constructor
- hardcoded values for the video source, output format, and encoder format

Of particular interest is the call to `setVideoSource()`. Usually, you would set this to `CAMERA`, to record from a device-supplied camera. Here, though, we set it to `SURFACE`, indicating that `MediaRecorder` should supply a `Surface` onto which we can render what should get recorded.

We then:

- Prepare the `MediaRecorder`, which might throw an `IOException` if there is some problem with the output file
- Create a `VirtualDisplay`, as we did in andprojector, tied to the details of the display we got from `DisplayMetrics` by way of the `RecordingConfig`
- Play a tone using `ToneGenerator` to let the user know that recording has begun
- Actually begin the recording, via a call to `start()` on the `MediaRecorder`

The `VIRT_DISPLAY_FLAGS` used here are the same ones used for andprojector:

```
  static final int VIRT_DISPLAY_FLAGS=
    DisplayManager.VIRTUAL_DISPLAY_FLAG_OWN_CONTENT_ONLY |
      DisplayManager.VIRTUAL_DISPLAY_FLAG_PUBLIC;
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingSession.java)

And, at this point, the screen is being recorded.

Stopping the Recording

Eventually, we will want to stop that recording, whether triggered via the `Notification` or the command-line script. That eventually results in a call to `stopRecorder()` on the `RecorderService`, which just calls `stop` on the `RecordingSession` before setting the field to `null`:

```
synchronized private void stopRecorder() {
  if (session!=null) {
    session.stop();
    session=null;
  }
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecorderService.java)

The `stop()` method on `RecordingSession` unwinds everything we set up, via `stop()` and `release()` calls on the `MediaProjection`, `MediaRecorder`, and `VirtualDisplay`. `stop()` also calls `scanFile()` on `MediaScannerConnection`, so that our video gets indexed by the `MediaStore` and therefore can be seen in on-device video players and via the MTP connection to your developer machine:

```
void stop() {
  projection.stop();
  recorder.stop();
  recorder.release();
  vdisplay.release();

  MediaScannerConnection.scanFile(ctxt,
    new String[]{output.getAbsolutePath()}, null, this);
}

@Override
public void onScanCompleted(String path, Uri uri) {
  beeper.startTone(ToneGenerator.TONE_PROP_NACK);
}
```

(from MediaProjection/andcorder/app/src/main/java/com/commonsware/android/andcorder/RecordingSession.java)

When the scan is complete, another beep signals to the user that the screen recording is finished.

Usage Notes

On the plus side, andcorder has no built-in duration limitation, the way that `adb shell screenrecord` does.

However, it does not optimize configuration changes. If you rotate the device during the recording, the recording will continue, but the screen will be shrunk to fit within the original dimensions. So, for example, if you start recording in landscape, then rotate the device to portrait, the video will still be landscape, with part of the video showing a small portrait rendition of the screen.

Yet Another Sample: andshooter

`andprojector` offers automated screenshots, in the context of streaming those images to another device via a built-in Web server. `andcorder` offers screencast recording on-demand.

`andshooter` is a bit of a mash-up of those two: offering manual screenshots, driven from a `Notification` or, more commonly, from the command line.

The Activity

The `MainActivity` exists mostly as a way to get the media projection permission from the user, plus a way to start the `ScreenshotService` when that permission has been obtained:

```
package com.commonsware.android.andshooter;

import android.app.Activity;
import android.content.Intent;
import android.media.projection.MediaProjectionManager;
import android.os.Bundle;

public class MainActivity extends Activity {
  private static final int REQUEST_SCREENSHOT=59706;
  private MediaProjectionManager mgr;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mgr=(MediaProjectionManager)getSystemService(MEDIA_PROJECTION_SERVICE);

    startActivityForResult(mgr.createScreenCaptureIntent(),
      REQUEST_SCREENSHOT);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode==REQUEST_SCREENSHOT) {
      if (resultCode==RESULT_OK) {
        Intent i=
            new Intent(this, ScreenshotService.class)
                .putExtra(ScreenshotService.EXTRA_RESULT_CODE, resultCode)
                .putExtra(ScreenshotService.EXTRA_RESULT_INTENT, data);

        startService(i);
      }
    }

    finish();
  }
}
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/MainActivity.java)

It uses `Theme.Translucent.NoTitleBar` to have no UI, instead delegating the UI to Android's media projection permission dialog by means of `createScreenCaptureIntent()` on `MediaProjectionManager`. If the user grants permission, we pass the result code and result `Intent` along to the `ScreenshotService` in `onActivityResult()` via `startService()`.

Setting Up the Notification

The `onCreate()` method of the `ScreenshotService` simply obtains access to the `WindowManager` and `MediaProjectionManager` system services, plus sets up a `HandlerThread` for use with `MediaProjectionManager`:

```
@Override
public void onCreate() {
  super.onCreate();

  mgr=(MediaProjectionManager)getSystemService(MEDIA_PROJECTION_SERVICE);
  wmgr=(WindowManager)getSystemService(WINDOW_SERVICE);

  handlerThread.start();
  handler=new Handler(handlerThread.getLooper());
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ScreenshotService.java)

`onStartCommand()` performs different work based upon the `Intent` action used to start this service:

```
@Override
public int onStartCommand(Intent i, int flags, int startId) {
  if (i.getAction()==null) {
    resultCode=i.getIntExtra(EXTRA_RESULT_CODE, 1337);
    resultData=i.getParcelableExtra(EXTRA_RESULT_INTENT);
    foregroundify();
 }
  else if (ACTION_RECORD.equals(i.getAction())) {
    if (resultData!=null) {
      startCapture();
    }
    else {
      Intent ui=
        new Intent(this, MainActivity.class)
          .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

      startActivity(ui);
```

```
    }
  }
  else if (ACTION_SHUTDOWN.equals(i.getAction())) {
    beeper.startTone(ToneGenerator.TONE_PROP_NACK);
    stopForeground(true);
    stopSelf();
  }

  return(START_NOT_STICKY);
}
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ScreenshotService.java)

In the case where there is no action string, this must be the `Intent` from the `startService()` call in `MainActivity`. So, we grab and hold onto that result code and result `Intent` passed in via extras, then call `foregroundify()` to set up a the service as a foreground service with an associated `Notification`:

```
private void foregroundify() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this);

  b.setAutoCancel(true)
    .setDefaults(Notification.DEFAULT_ALL);

  b.setContentTitle(getString(R.string.app_name))
    .setSmallIcon(R.mipmap.ic_launcher)
    .setTicker(getString(R.string.app_name));

  b.addAction(R.drawable.ic_record_white_24dp,
    getString(R.string.notify_record),
    buildPendingIntent(ACTION_RECORD));

  b.addAction(R.drawable.ic_eject_white_24dp,
    getString(R.string.notify_shutdown),
    buildPendingIntent(ACTION_SHUTDOWN));

  startForeground(NOTIFY_ID, b.build());
}

private PendingIntent buildPendingIntent(String action) {
  Intent i=new Intent(this, getClass());

  i.setAction(action);

  return(PendingIntent.getService(this, 0, i, 0));
}
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ScreenshotService.java)

That `Notification`, in turn, has "record" and "shutdown" actions that will trigger `ScreenshotService` with custom action strings, to trigger other branches within `onStartCommand()`.

However, in truth, the expectation is that the user will trigger a screenshot from a shell script, so as not to disturb the foreground UI. The `record` shell script in the project root will take a screenshot, pull the file down from the device or emulator to a designated location, then remove the screenshot from the device or emulator:

```
#!/bin/bash

adb shell am startservice -n com.commonsware.android.andshooter/.ScreenshotService \
-a com.commonsware.android.andshooter.RECORD
sleep 2s

adb pull /storage/emulated/0/Android/data/com.commonsware.android.andshooter/files/
screenshot.png $1
adb shell rm /storage/emulated/0/Android/data/com.commonsware.android.andshooter/
files/screenshot.png
```

NOTE: the backslashes indicate where a line break was added to allow these lines to fit on the width of the page

Capturing a Screenshot

If the user asks to record a screenshot — via the `Notification` or the shell script — `captureImage()` is called:

```
  private void startCapture() {
    projection=mgr.getMediaProjection(resultCode, resultData);
    it=new ImageTransmogrifier(this);

    MediaProjection.Callback cb=new MediaProjection.Callback() {
      @Override
      public void onStop() {
        vdisplay.release();
      }
    };

    vdisplay=projection.createVirtualDisplay("andshooter",
      it.getWidth(), it.getHeight(),
      getResources().getDisplayMetrics().densityDpi,
      VIRT_DISPLAY_FLAGS, it.getSurface(), null, handler);
```

```
    projection.registerCallback(cb, handler);
  }
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ScreenshotService.java)

Here we:

- Get a `MediaProjection`
- Create an `ImageTransmogrifier`
- Create a virtual display corresponding to our screen size
- Register a callback to find out when we are done with the `MediaProjection`, so we can also get rid of the virtual display at the same time

`ImageTransmogrifier` is largely the same as in `andprojector`, arranging to get the images as they become available and creating a PNG out of them:

```
package com.commonsware.android.andshooter;

import android.graphics.Bitmap;
import android.graphics.PixelFormat;
import android.graphics.Point;
import android.media.Image;
import android.media.ImageReader;
import android.view.Display;
import android.view.Surface;
import java.io.ByteArrayOutputStream;
import java.nio.ByteBuffer;

public class ImageTransmogrifier implements ImageReader.OnImageAvailableListener {
  private final int width;
  private final int height;
  private final ImageReader imageReader;
  private final ScreenshotService svc;
  private Bitmap latestBitmap=null;

  ImageTransmogrifier(ScreenshotService svc) {
    this.svc=svc;

    Display display=svc.getWindowManager().getDefaultDisplay();
    Point size=new Point();

    display.getSize(size);

    int width=size.x;
    int height=size.y;

    while (width*height > (2<<19)) {
      width=width>>1;
      height=height>>1;
    }

    this.width=width;
    this.height=height;
```

```
  imageReader=ImageReader.newInstance(width, height,
      PixelFormat.RGBA_8888, 2);
  imageReader.setOnImageAvailableListener(this, svc.getHandler());
}

@Override
public void onImageAvailable(ImageReader reader) {
  final Image image=imageReader.acquireLatestImage();

  if (image!=null) {
    Image.Plane[] planes=image.getPlanes();
    ByteBuffer buffer=planes[0].getBuffer();
    int pixelStride=planes[0].getPixelStride();
    int rowStride=planes[0].getRowStride();
    int rowPadding=rowStride - pixelStride * width;
    int bitmapWidth=width + rowPadding / pixelStride;

    if (latestBitmap == null ||
        latestBitmap.getWidth() != bitmapWidth ||
        latestBitmap.getHeight() != height) {
      if (latestBitmap != null) {
        latestBitmap.recycle();
      }

      latestBitmap=Bitmap.createBitmap(bitmapWidth,
          height, Bitmap.Config.ARGB_8888);
    }

    latestBitmap.copyPixelsFromBuffer(buffer);

    if (image != null) {
      image.close();
    }

    ByteArrayOutputStream baos=new ByteArrayOutputStream();
    Bitmap cropped=Bitmap.createBitmap(latestBitmap, 0, 0,
      width, height);

    cropped.compress(Bitmap.CompressFormat.PNG, 100, baos);

    byte[] newPng=baos.toByteArray();

    svc.processImage(newPng);
  }
}

Surface getSurface() {
  return(imageReader.getSurface());
}

int getWidth() {
  return(width);
}

int getHeight() {
  return(height);
}

void close() {
  imageReader.close();
```

```
  }
}
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ImageTransmogrifier.java)

That, in turn, triggers a call to `processImage()`, where we write the PNG to a file, update the `MediaStore` so it knows about the image, play an acknowledgment tone to let the user know the screenshot is ready, and call `stopCapture()`:

```
void processImage(final byte[] png) {
  new Thread() {
    @Override
    public void run() {
      File output=new File(getExternalFilesDir(null),
        "screenshot.png");

      try {
        FileOutputStream fos=new FileOutputStream(output);

        fos.write(png);
        fos.flush();
        fos.getFD().sync();
        fos.close();

        MediaScannerConnection.scanFile(ScreenshotService.this,
          new String[] {output.getAbsolutePath()},
          new String[] {"image/png"},
          null);
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(), "Exception writing out screenshot", e);
      }
    }
  }.start();

  beeper.startTone(ToneGenerator.TONE_PROP_ACK);
  stopCapture();
}
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ScreenshotService.java)

All `stopCapture()` does is close down the `MediaProjection` and associated virtual display, to clean things up in preparation for the next screenshot:

```
private void stopCapture() {
  if (projection!=null) {
    projection.stop();
    vdisplay.release();
    projection=null;
  }
}
```

(from MediaProjection/andshooter/app/src/main/java/com/commonsware/android/andshooter/ScreenshotService.java)

AlarmManager and the Scheduled Service Pattern

Many applications have the need to get control every so often to do a bit of work. And, many times, those applications need to get control in the background, regardless of what the user may be doing (or not doing) at the time.

The solution, in some cases, is to use `AlarmManager`, which is roughly akin to `cron` on Linux and macOS and Scheduled Tasks in Windows. You teach `AlarmManager` when you want to get control back, and `AlarmManager` will give you control at that time.

Android 5.0 added a separate `JobScheduler`. Like `AlarmManager`, `JobScheduler` is designed for background work. `JobScheduler` is more sophisticated than is `AlarmManager`. For example, if you need an Internet connection to do your work, `JobScheduler` will only give you control if there is an Internet connection. If your app's `minSdkVersion` is 21 or higher, you might consider using `JobScheduler` instead of `AlarmManager`. `JobScheduler` is covered in an upcoming chapter.

Scenarios

The two main axes to consider with scheduled work are frequency and foreground (vs. background).

If you have an activity that needs to get control every second, the simplest approach is to use a `postDelayed()` loop, scheduling a `Runnable` to be invoked after a certain delay, where the `Runnable` reschedules itself to be invoked after the delay in addition to doing some work. We saw this in the chapter on threads. This has the advantages

of giving you control back on the main application thread and avoiding the need for any background threads.

On the far other end of the spectrum, you may need to get control on a somewhat slower frequency (e.g., every 15 minutes), and do so in the background, even if nothing of your app is presently running. You might need to poll some Web server for new information, such as downloading updates to an RSS feed. This is the scenario that `AlarmManager` excels at. While `postDelayed()` works *inside* your process (and therefore does not work if you no longer have a process), `AlarmManager` maintains its schedule *outside* of your process. Hence, it can arrange to give you control, even if it has to start up a new process for you along the way.

Options

There are a variety of things you will be able to configure about your scheduled alarms with `AlarmManager`.

Wake Up… Or Not?

The biggest one is whether or not the scheduled event should wake up the device.

A device goes into a sleep mode shortly after the screen goes dark. During this time, nothing at the application layer will run, until something wakes up the device. Waking up the device does not necessarily turn on the screen — it may just be that the CPU starts running your process again.

If you choose a "wakeup"-style alarm, Android will wake up the device to give you control. This would be appropriate if you need this work to occur even if the user is not actively using the device, such as your app checking for critical email messages in the middle of the night. However, it does drain the battery some.

Alternatively, you can choose an alarm that will not wake up the device. If your desired time arrives and the device is asleep, you will not get control until something else wakes up the device.

Repeating… Or Not?

You can create a "one-shot" alarm, to get control once at a particular time in the future. Or, you can create an alarm that will give you control periodically, at a fixed period of your choice (e.g., every 15 minutes).

If you need to get control at multiple times, but the schedule is irregular, use a "one-shot" alarm for the nearest time, where you do your work *and* schedule a "one-shot" alarm for the next-nearest time. This would be appropriate for scenarios like a calendar application, where you need to let the user know about upcoming appointments, but the times for those appointments may not have any fixed schedule.

However, for most polling operations (e.g., checking for new messages every NN minutes), a repeating alarm will typically be the better answer.

Inexact… Or Not?

If you do choose a repeating alarm, you will have your choice over having (relatively) precise control over the timing of event or not.

If you choose an "inexact" alarm, while you will provide Android with a suggested time for the first event and a period for subsequent events, Android reserves the right to shift your schedule somewhat, so it can process your events and others around the same time. This is particularly important for "wakeup"-style alarms, as it is more power-efficient to wake up the device fewer times, so Android will try to combine multiple apps' events to be around the same time to minimize the frequency of waking up the device.

However, inexact alarms are annoying to test and debug, simply because you do not have control over when they will be invoked. Hence, during development, you might start with an exact alarm, then switch to inexact alarms once most of your business logic is debugged.

Note that Android 4.4 changes the behavior of `AlarmManager`, such that it is more difficult to actually create an exact-repeating alarm schedule. This will be examined in greater detail shortly, as we review the various methods and flags for scheduling `AlarmManager` events.

Absolute Time… Or Not?

As part of the alarm configuration, you will tell Android when the event is to occur (for one-shot alarms) or when the event is to *first* occur (for repeating alarms). You can provide that time in one of two ways:

- An absolute "real-time clock" time (e.g., 4am tomorrow), or
- A time relative to now

For most polling operations, particularly for periods more frequent than once per day, specifying the time relative to now is easiest. However, some alarms may need to tie into "real world time", such as alarm clocks and calendar alerts — for those, you will need to use the real-time clock (typically by means of a Java `Calendar` object) to indicate when the event should occur.

What Happens (Or Not???)

And, of course, you will need to tell Android what to do when each of these timer events occurs. You will do that in the form of supplying a `PendingIntent`. First mentioned in the chapter on services, a `PendingIntent` is a `Parcelable` object, one that indicates an operation to be performed upon an `Intent`:

- start an activity
- start a service
- send a broadcast

While the service chapter discussed an Android activity using `createPendingResult()` to craft such a `PendingIntent`, that is usually not very useful for `AlarmManager`, as the `PendingIntent` will only be valid so long as the activity is in the foreground. Instead, there are static factory methods on `PendingIntent` that you will use instead (e.g., `getBroadcast()` to create a `PendingIntent` that calls `sendBroadcast()` on a supplied `Intent`). That being said, our next sample will use `createPendingResult()`, to keep the sample as simple as possible.

A Simple Example

A trivial sample app using `AlarmManager` can be found in `AlarmManager/Simple`.

This application consists of a single activity, `SimpleAlarmDemoActivity`, that will both set up an alarm schedule and respond to alarms:

```
package com.commonsware.android.alarm;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.os.SystemClock;
```

```
import android.util.Log;
import android.widget.Toast;

public class SimpleAlarmDemoActivity extends Activity {
  private static final int ALARM_ID=1337;
  private static final int PERIOD=5000;
  private PendingIntent pi=null;
  private AlarmManager mgr=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mgr=(AlarmManager)getSystemService(ALARM_SERVICE);
    pi=createPendingResult(ALARM_ID, new Intent(), 0);
    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME,
                     SystemClock.elapsedRealtime() + PERIOD, PERIOD, pi);
  }

  @Override
  public void onDestroy() {
    mgr.cancel(pi);

    super.onDestroy();
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == ALARM_ID) {
      Toast.makeText(this, R.string.toast, Toast.LENGTH_SHORT).show();
      Log.d(getClass().getSimpleName(), "I ran!");
    }
  }
}
```

(from AlarmManager/Simple/app/src/main/java/com/commonsware/android/alarm/SimpleAlarmDemoActivity.java)

In `onCreate()`, in addition to setting up the "hello, world"-ish UI, we:

- Obtain an instance of `AlarmManager`, by calling `getSystemService()`, asking for the `ALARM_SERVICE`, and casting the result to be an `AlarmManager`
- Create a `PendingIntent` by calling `createPendingResult()`, supplying an empty `Intent` as our "result" (since we do not really need it here)
- Calling `setRepeating()` on `AlarmManager`

The call to `setRepeating()` is a bit complex, taking four parameters:

1. The type of alarm we want, in this case `ELAPSED_REALTIME`, indicating that we want to use a relative time base for when the first event should occur (i.e., relative to now) and that we do not need to wake up the device out of any sleep mode
2. The time when we want the first event to occur, in this case specified as a time delta in milliseconds (`PERIOD`) added to "now" as determined by `SystemClock.elapsedRealtime()` (the number of milliseconds since the device was last rebooted)
3. The number of milliseconds to occur between events
4. The `PendingIntent` to invoke for each of these events

When the event occurs, since we used `createPendingResult()` to create the `PendingIntent`, our activity gets control in `onActivityResult()`, where we simply display a `Toast` (if the event is for our alarm's request ID). This continues until the activity is destroyed (e.g., pressing the BACK button), at which time we `cancel()` the alarm, supplying a `PendingIntent` to indicate which alarm to cancel. While here we use the same `PendingIntent` object as we used for scheduling the alarm, that is not required — it merely has to be an *equivalent* `PendingIntent`, meaning:

- The `Intent` inside the `PendingIntent` matches the scheduled alarm's `Intent`, in terms of component, action, data (`Uri`), MIME type, and categories
- The ID of the `PendingIntent` (here, `ALARM_ID`) must also match

Running this simply brings up a `Toast` every five seconds until you BACK out of the activity.

The Five set…() Varieties

There are five methods that you can call on `AlarmManager` to establish an alarm, including the `setRepeating()` demonstrated above.

On Android 4.4 (API Level 19) and higher, `setExact()` is used for a one-shot alarm, where you want to get control at one specific time in the future. This would be used for specific events or for irregular alarm schedules.

On Android 4.3 and below, and for apps whose `targetSdkVersion` is set to 18 or lower, `set()` has the same behavior as `setExact()`. However, on Android 4.4 and above, apps with their `targetSdkVersion` set to be 19 or higher will have different,

inexact behavior for `set()`. The time of the event is considered a minimum — your `PendingIntent` will not be invoked before your desired time, but it can occur any time thereafter... and you do not have control over how long that delay will be. As with all "inexact" schedules, the objective is for Android to be able to "batch" these events, to do several around the same time, for greater efficiency, particularly when waking up the device.

On Android 4.4 and higher, you have a `setWindow()` option that is a bit of a hybrid between the new-style `set()` and `setExact()`. Here, you specify the time you want the event to occur and an amount of time that Android can "flex" the actual event. So, for example, you might set up an event to occur every hour, with a "window" of five minutes, to allow Android the flexibility to invoke your `PendingIntent` within that five-minute window. This allows for better battery optimization than with `setExact()`, while still giving you some control over how far "off the mark" the event can occur.

On Android 4.3 and below, and for apps whose `targetSdkVersion` is set to 18 or lower, `setRepeating()` is used for an alarm that should occur at specific points in time at a specific frequency. In addition to specifying the time of the first event, you also specify the period for future events. Android will endeavor to give you control at precisely those times, though since Android is not a real-time operating system (RTOS), microsecond-level accuracy is certainly not guaranteed. However, note that as of Android 5.1, your minimum period is one minute (60000ms) — values less than that will be rounded up to one minute. This minimum period is enforced regardless of your `targetSdkVersion` value.

`setInexactRepeating()` is used for an alarm that should occur on a general frequency, such as every 15 minutes. In addition to specifying the time of the first event, you also specify a general frequency, as one of the following public static data members on `AlarmManager`:

- `INTERVAL_FIFTEEN_MINUTES`
- `INTERVAL_HALF_HOUR`
- `INTERVAL_HOUR`
- `INTERVAL_HALF_DAY`
- `INTERVAL_DAY`

Android guarantees that it will give your app control somewhere during that time window, but precisely *when* within that window is up to Android.

Note that on Android 4.4 and above, for apps with their `targetSdkVersion` set to be 19 or higher, `setRepeating()` behaves identically to `setInexactRepeating()` – in other words, all repeating alarms are inexact. The only way to get exact repeating would be to use `setExact()` and to re-schedule the event yourself, rather than relying upon Android doing that for you automatically. Ideally, you use `setInexactRepeating()`, to help extend battery life.

And, note that on Android 5.1 and higher, alarms must be set to occur at least 5 seconds in the future from now. You cannot trigger an alarm to occur in the future sooner than 5 seconds.

The Four Types of Alarms

In the above sample, we used `ELAPSED_REALTIME` as the type of alarm. There are three others:

- `ELAPSED_REALTIME_WAKEUP`
- `RTC`
- `RTC_WAKEUP`

Those with `_WAKEUP` at the end will wake up a device out of sleep mode to execute the `PendingIntent` — otherwise, the alarm will wait until the device is awake for other means.

Those that begin with `ELAPSED_REALTIME` expect the second parameter to `setRepeating()` to be a timestamp based upon `SystemClock.elapsedRealtime()`. Those that begin with `RTC`, however, expect the second parameter to be based upon `System.currentTimeMillis()`, the classic Java "what is the current time in milliseconds since the Unix epoch" method.

When to Schedule Alarms

The sample, though, begs a bit of a question: when are we supposed to set up these alarms? The sample just does so in `onCreate()`, but is that sufficient?

For most apps, the answer is "no". Here are the three times that you will need to ensure that your alarms get scheduled:

When User First Runs Your App

When your app is first installed, none of your alarms are set up, because your code has not yet run to schedule them. There is no means of setting up alarm information in the manifest or something that might automatically kick in.

Hence, you will need to schedule your alarms when the user first runs your app.

As a simplifying measure — and to cover another scenario outlined below — you might be able to simply get away with scheduling your alarms *every* time the user runs your app, as the sample app shown above does. This works for one-shot alarms (using `set()`) and for alarms with short polling periods, and it works because setting up a new alarm schedule for an equivalent `PendingIntent` will replace the old schedule. However, for repeating alarms with slower polling periods, it may excessively delay your events. For example, suppose you have an alarm set to go off every 24 hours, and the user happens to run your app 5 minutes before the next event was to occur — if you blindly reschedule the alarm, instead of going off in 5 minutes, it might not go off for another 24 hours.

There are more sophisticated approaches for this (e.g., using a `SharedPreferences` value to determine if your app has run before or not).

On Boot

The alarm schedule for alarm manager is wiped clean on a reboot, unlike `cron` or Windows Scheduled Tasks. Hence, you will need to get control at boot time to re-establish your alarms, if you want them to start up again after a reboot. We saw how to get control at boot time, via an `ACTION_BOOT_COMPLETED` `BroadcastReceiver`, back in the chapter on broadcasts.

After a Force-Stop

There are other events that could cause your alarms to become unscheduled. The best example of this is if the user goes into the Settings app and presses "Force Stop" for your app. At this point, on Android 3.1+, nothing of your code will run again, until the user manually launches some activity of yours.

If you are rescheduling your alarms every time your app runs, this will be corrected the next time the user launches your app. And, by definition, you cannot do anything until the user runs one of your activities, anyway.

If you are trying to avoid rescheduling your alarms on each run, though, you have a couple of options.

One is to record the time when your alarm-triggered events occur, each time they occur, such as by updating a `SharedPreference`. When the user launches one of your activities, you check the last-event time — if it was too long ago (e.g., well over your polling period), you assume that the alarm had been canceled, and you reschedule it.

Another is to rely on `FLAG_NO_CREATE`. You can pass this as a parameter to any of the `PendingIntent` factory methods, to indicate that Android should only return an *existing* `PendingIntent` if there is one, and not create one if there is not:

```
PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, PendingIntent.FLAG_NO_CREATE);
```

If the `PendingIntent` is `null`, your alarm has been canceled — otherwise, Android would already have such a `PendingIntent` and would have returned it to you. This feels a bit like a side-effect, so we cannot rule out the possibility that, in future versions of Android, this technique could result in false positives (`null` `PendingIntent` despite the scheduled alarm) or false negatives (non-`null` `PendingIntent` despite a canceled alarm).

Archetype: Scheduled Service Polling

The classic `AlarmManager` scenario is where you want to do a chunk of work, in the background, on a periodic basis. This is fairly simple to set up in Android, though perhaps not quite as simple as you might think.

Back to the Main Application Thread

When an `AlarmManager`-triggered event occurs, it is very likely that your application is not running. This means that the `PendingIntent` is going to have to start up your process to have you do some work. Since everything that a `PendingIntent` can do intrinsically gives you control on your main application thread, you are going to have to determine how you want to move your work to a background thread.

One approach is to use a `PendingIntent` created by `getService()`, and have it send a command to an `IntentService` that you write. Since `IntentService` does its work on a background thread, you can take whatever time you need, without interfering

with the behavior of the main application thread. This is particularly important when:

- The `AlarmManager`-triggered event happens to occur when the user happens to have one of your activities in the foreground, so you do not freeze the UI, or
- You want the same business logic to be executed on demand by the user, such as via an action bar item, as once again you do not want to freeze the UI

Problem: Keeping the Device Awake

However, this approach has a flaw: the device might fall asleep before our service can complete its work, if we woke it up out of sleep mode to process the event.

For a `_WAKEUP`-style alarm, Android makes precisely one guarantee: *if* the `PendingIntent` supplied to `AlarmManager` for the alarm is one created by `getBroadcast()` to send a broadcast `Intent`, Android will ensure that the device will stay awake long enough for `onReceive()` to be completed. Anything beyond that is not guaranteed.

While sending the command directly to the service via a `getService()` `PendingIntent` is straightforward, Android makes no guarantees about what happens after `AlarmManager` wakes up the device, and the device could fall back asleep before our `IntentService` completes processing of `onHandleIntent()`.

So, we have two problems:

1. How do we ensure that the service itself keeps the device awake while it is doing its work?
2. How do we ensure that we can actually get that service started, without the device falling asleep after we return from `onReceive()`?

Return of the JobIntentService

`JobIntentService` neatly solves both of those problems. It takes responsibility for keeping the device awake, both to set up the work, and to process the work.

So, what we need to do is put our business logic in a `JobIntentService`, then use a broadcast `PendingIntent` to trigger enqueuing work to that `JobIntentService`. `JobIntentService` will take it from there.

Examining a Sample

An incrementally-less-trivial sample app using `AlarmManager` for the scheduled service pattern can be found in `AlarmManager/Scheduled`.

This application consists of three components: a `BroadcastReceiver`, a `JobIntentService`, and an `Activity`.

This sample demonstrates scheduling your alarms at two points in your app:

- At boot time
- When the user runs the activity

For the boot-time scenario, we need a `BroadcastReceiver` set up to receive the `ACTION_BOOT_COMPLETED` broadcast, with the appropriate permission. So, we set that up, along with our other components, in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.wakesvc"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.WAKE_LOCK" />
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="ScheduledServiceDemoActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <receiver android:name="PollReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
    </receiver>
```

```
    <service
      android:name="ScheduledService"
      android:permission="android.permission.BIND_JOB_SERVICE" />
  </application>

</manifest>
```

(from AlarmManager/Scheduled/app/src/main/AndroidManifest.xml)

The `PollReceiver` has its `onReceive()` method, to be called at boot time, which delegates its work to a `scheduleAlarms()` static method, so that logic can also be used by our activity:

```
package com.commonsware.android.wakesvc;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class PollReceiver extends BroadcastReceiver {
  private static final int PERIOD=900000; // 15 minutes
  private static final int INITIAL_DELAY=5000; // 5 seconds

  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() == null) {
      ScheduledService.enqueueWork(ctxt);
    }
    else {
      scheduleAlarms(ctxt);
    }
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime() + INITIAL_DELAY,
                     PERIOD, pi);
```

```
  }
}
```

(from AlarmManager/Scheduled/app/src/main/java/com/commonsware/android/wakesvc/PollReceiver.java)

The `scheduleAlarms()` method retrieves our `AlarmManager`, creates a `PendingIntent` designed to call `startService()` on our `ScheduledService`, and schedules an exact repeating alarm to have that command be sent every five seconds.

The `ScheduledService` itself is the epitome of "trivial", simply logging a message to Logcat on each command:

```
package com.commonsware.android.wakesvc;

import android.content.Context;
import android.content.Intent;
import android.support.v4.app.JobIntentService;
import android.util.Log;

public class ScheduledService extends JobIntentService {
  private static final int UNIQUE_JOB_ID=1337;

  static void enqueueWork(Context ctxt) {
    enqueueWork(ctxt, ScheduledService.class, UNIQUE_JOB_ID,
      new Intent(ctxt, ScheduledService.class));
  }

  @Override
  public void onHandleWork(Intent i) {
    Log.d(getClass().getSimpleName(), "I ran!");
  }
}
```

(from AlarmManager/Scheduled/app/src/main/java/com/commonsware/android/wakesvc/ScheduledService.java)

That being said, because this is an `JobIntentService`, we could do much more in `onHandleWork()` and not worry about tying up the main application thread.

Our activity — `ScheduledServiceDemoActivity` — is set up with `Theme.Translucent.NoTitleBar` in the manifest, never calls `setContentView()`, and calls `finish()` right from `onCreate()`. As a result, it has no UI. It simply calls `scheduleAlarms()` and raises a `Toast` to indicate that the alarms are indeed scheduled:

```
package com.commonsware.android.wakesvc;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;

public class ScheduledServiceDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PollReceiver.scheduleAlarms(this);

    Toast.makeText(this, R.string.alarms_scheduled, Toast.LENGTH_LONG)
         .show();
    finish();
  }
}
```

(from AlarmManager/Scheduled/app/src/main/java/com/commonsware/android/wakesvc/ScheduledServiceDemoActivity.java)

On Android 3.1+, we also need this activity to move our application out of the stopped state and allow that boot-time `BroadcastReceiver` to work.

If you run this app on a device or emulator, after seeing the initial `Toast`, messages will appear in Logcat every five seconds, even though you have no activity running.

How the Magic Works

A `JobIntentService` keeps the device awake by using a `WakeLock`. A `WakeLock` allows a "userland" (e.g., Android SDK) app to tell the Linux kernel at the heart of Android to keep the device awake, with the CPU powered on, indefinitely, until the `WakeLock` is released.

This can be a wee bit dangerous, as you can accidentally keep the device awake much longer than you need to. That is why using existing code like `JobIntentService` can be useful — to use more-tested code rather than rolling your own.

Also note that to request a `WakeLock` — from your code or code from a library – you need to request the `WAKE_LOCK` permission, as we saw in the manifest:

```
  <uses-permission android:name="android.permission.WAKE_LOCK" />
```

(from AlarmManager/Scheduled/app/src/main/AndroidManifest.xml)

Warning: Not All Android Devices Play Nice

Some Android devices take liberties with the way `AlarmManager` works, in ways that may affect your applications.

One example of this today is the SONY Xperia Z. It has a "STAMINA mode" that the user can toggle on via the "Power Management" screen in Settings. This mode will be entered when the device's screen turns off, if the device is not plugged in and charging. The user can add apps to a whitelist ("Apps active in standby"), where STAMINA mode does not affect those apps' behavior.

`_WAKEUP` style alarms do not wake up the device when it is in STAMINA mode. The behavior is a bit reminiscent of non-`_WAKEUP` alarms. Alarms that occur while the device is asleep are suppressed, and you get one invocation of your `PendingIntent` at the point the device wakes back up. At that point, the schedule continues as though the alarms had been going off all along. Apps on the whitelist are unaffected.

Mostly, you need to be aware of this from a support standpoint. If Xperia Z owners complain that your app behaves oddly, and you determine that your alarms are not going off, see if they have STAMINA mode on, and if they do, ask them to add your app to the whitelist.

If you are using "if my alarm has not gone off in X amount of time, the user perhaps force-stopped me, so let me reschedule my alarms" logic, you should be OK. Before one of your activities gets a chance to make that check, your post-wakeup alarm should have been invoked, so you can update your event log and last-run timestamp. Hence, you should not be tripped up by STAMINA and accidentally reschedule your alarms (potentially causing duplicates, depending upon your alarm-scheduling logic).

Other devices with similar characteristics include Sony's Xperia P, Xperia U, Xperia sola, and Xperia go.

Debugging Alarms

If you are encountering issues with your alarms, the first thing to do is to ensure that the alarm schedule in `AlarmManager` is what you expect it to be. To do that, run `adb shell dumpsys alarm` from a command prompt. This will dump a report of all the scheduled alarms, including when they are set to be invoked next (with portions replaced by vertical ellipses to keep this listing from being too long):

```
Current Alarm Manager state:

  Realtime wakeup (now=2013-03-09 07:49:51):
  RTC_WAKEUP #11: Alarm{429c6028 type 0 com.android.providers.calendar}
    type=0 when=+21h40m9s528ms repeatInterval=0 count=0
    operation=PendingIntent{42ec2f40: PendingIntentRecord{434fb2f8
com.android.providers.calendar broadcastIntent}}
  RTC_WAKEUP #10: Alarm{42e17e28 type 0 com.google.android.gms}
    type=0 when=+18h10m8s480ms repeatInterval=86400000 count=1
    operation=PendingIntent{42e15d20: PendingIntentRecord{42e0cc28
com.google.android.gms startService}}
.
.
.

  Elapsed realtime wakeup (now=+6d15h50m2s672ms):
  ELAPSED_WAKEUP #16: Alarm{42cf26f0 type 2 com.google.android.apps.maps}
    type=2 when=+999d23h59m59s999ms repeatInterval=0 count=0
    operation=PendingIntent{42de2dc0: PendingIntentRecord{42ac73e8
com.google.android.apps.maps broadcastIntent}}
  ELAPSED_WAKEUP #15: Alarm{42c4a638 type 2 com.google.android.apps.maps}
    type=2 when=+1d18h10m8s894ms repeatInterval=0 count=0
    operation=PendingIntent{42ab50c8: PendingIntentRecord{42e2c020
com.google.android.apps.maps broadcastIntent}}
.
.
.

  Broadcast ref count: 0

  Top Alarms:
    +14m24s97ms running, 0 wakeups, 9567 alarms: android
       act=android.intent.action.TIME_TICK
    +1m15s72ms running, 4890 wakeups, 4890 alarms: com.android.phone
       act=com.android.server.sip.SipWakeupTimer@42626830
    +1m13s465ms running, 0 wakeups, 320 alarms: android
       act=com.android.server.action.NETWORK_STATS_POLL
    +45s803ms running, 0 wakeups, 639 alarms: com.google.android.deskclock
       act=com.android.deskclock.ON_QUARTER_HOUR
    +42s830ms running, 0 wakeups, 19 alarms: com.android.phone
       act=com.android.phone.UPDATE_CALLER_INFO_CACHE cmp={com.android.phone/
com.android.phone.CallerInfoCacheUpdateReceiver}
    +35s479ms running, 0 wakeups, 954 alarms: android
       act=com.android.server.ThrottleManager.action.POLL
    +14s28ms running, 1609 wakeups, 1609 alarms: com.android.phone
       act=com.android.internal.telephony.gprs-data-stall
    +11s98ms running, 171 wakeups, 171 alarms: com.android.providers.calendar
       act=com.android.providers.calendar.intent.CalendarProvider2
```

```
    +8s380ms running, 893 wakeups, 893 alarms: android
       act=android.content.syncmanager.SYNC_ALARM
    +8s353ms running, 569 wakeups, 569 alarms: com.google.android.apps.maps
       cmp={com.google.android.apps.maps/
com.google.googlenav.prefetch.android.PrefetcherService}

  Alarm Stats:
  com.google.android.location +120ms running, 12 wakeups:
    +73ms 7 wakes 7 alarms:
act=com.google.android.location.nlp.ALARM_WAKEUP_CACHE_UPDATER
    +47ms 5 wakes 5 alarms: act=com.google.android.location.nlp.ALARM_WAKEUP_LOCATOR
  android +15m32s920ms running, 1347 wakeups:
    +14m24s97ms 0 wakes 9567 alarms: act=android.intent.action.TIME_TICK
    +1m13s465ms 0 wakes 320 alarms: act=com.android.server.action.NETWORK_STATS_POLL
    +35s479ms 0 wakes 954 alarms: act=com.android.server.ThrottleManager.action.POLL
    +8s380ms 893 wakes 893 alarms: act=android.content.syncmanager.SYNC_ALARM
    +7s734ms 159 wakes 159 alarms: act=android.appwidget.action.APPWIDGET_UPDATE
cmp={com.guywmustang.silentwidget/
com.guywmustang.silentwidgetlib.SilentWidgetProvider}
    +1s144ms 151 wakes 151 alarms: act=android.app.backup.intent.RUN
    +922ms 0 wakes 6 alarms: act=android.intent.action.DATE_CHANGED
    +479ms 66 wakes 66 alarms: act=com.android.server.WifiManager.action.DEVICE_IDLE
    +383ms 56 wakes 56 alarms:
act=com.android.server.WifiManager.action.DELAYED_DRIVER_STOP
    +101ms 14 wakes 14 alarms: act=com.android.server.action.UPDATE_TWILIGHT_STATE
    +100ms 7 wakes 7 alarms:
act=com.android.internal.policy.impl.PhoneWindowManager.DELAYED_KEYGUARD
    +9ms 1 wakes 1 alarms: act=android.net.wifi.DHCP_RENEW
    +3ms 0 wakes 1 alarms: act=com.android.server.NetworkTimeUpdateService.action.POLL
  com.google.android.apps.maps +14s742ms running, 911 wakeups:
    +8s353ms 569 wakes 569 alarms: cmp={com.google.android.apps.maps/
com.google.googlenav.prefetch.android.PrefetcherService}
    +2s211ms 85 wakes 85 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_LOCATOR
    +1s206ms 103 wakes 103 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_SENSOR_UPLOADER
    +807ms 2 wakes 2 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_BURST_COLLECTION_TRIGGER
    +759ms 56 wakes 56 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_S_COLLECTOR
    +566ms 10 wakes 10 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_CACHE_UPDATER
    +385ms 39 wakes 39 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_IN_OUT_DOOR_COLLECTOR
    +308ms 31 wakes 31 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_ACTIVE_COLLECTOR
    +77ms 8 wakes 8 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_ACTIVITY_DETECTION
```

```
    +42ms 4 wakes 4 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_PASSIVE_COLLECTOR
    +28ms 4 wakes 4 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_CALIBRATION_COLLECTOR
.
.
.
```

You are given details of each outstanding alarm, including the all-important `when` value indicating the time the alarm should be invoked next, if it is not canceled first (e.g., `when=+5d15h10m7s782ms`), along with the package requesting the alarm. You can use this to identify your app's alarms and see when they should be invoked next.

You are also given:

- Per-app details about how frequently their alarms have gone off, which can be useful for battery impact analysis
- A list of "top alarms" by number of occurrences, also for device performance analysis

Note, though, that for inexact alarms, the `when` value may not indicate when the event will actually occur.

Android 6.0 and the War on Background Processing

Android 6.0 introduced some changes to the behavior of `AlarmManager` that significantly affect its use on Android 6.0+ devices. These changes also affect `JobScheduler`, and so this topic is covered in grand detail at the end of the `JobScheduler` chapter.

Android 7.0 and OnAlarmListener

Android 7.0 introduced a curious variant of the existing `set()`, `setExact()`, and `setWindow()` methods. Rather than taking a `PendingIntent`, they take an implementation of `OnAlarmListener`. That listener's `onAlarm()` method then gets called when the alarm is scheduled to go off.

These methods are only useful if the app has a process running and that process is likely to *be* running when the time for the alarm is to occur. If the app's process is

terminated after the `OnAlarmListener` is registered, the alarms are canceled, as the `OnAlarmListener` no longer exists.

For `RTC` and `ELAPSED_REALTIME` alarms, it is unclear what value there is in these `AlarmManager` methods over using some other in-process timing mechanism, such as Java's `ScheduledExecutorService`.

However, for `RTC_WAKEUP` and `ELAPSED_REALTIME_WAKEUP` alarms, the new `OnAlarmListener` methods may be useful, if you expect the device to be asleep but the process still running, and you want to get control to go do something. However, they still only make sense if you only want to get control *if* you already have a process running, and if your process goes away you do not mind the alarms going away.

There may be a set of apps that could use this. The author cannot quite figure out what such an app would be.

To illustrate the use of `OnAlarmListener`, we can turn to the `AlarmManager/Listener` sample app. This is reminiscent of the `AlarmManager/Simple` app shown in the beginning of this chapter, where we want an activity to get control every five seconds to show a `Toast` However, in this case, rather than use `setRepeating()` and `createPendingResult()`, we will use `setWindow()` and `OnAlarmListener`:

```
package com.commonsware.android.alarm;

import android.app.Activity;
import android.app.AlarmManager;
import android.os.Bundle;
import android.os.SystemClock;
import android.util.Log;
import android.widget.Toast;

public class SimpleAlarmDemoActivity extends Activity
  implements AlarmManager.OnAlarmListener {
  private static final int PERIOD=5000;
  private static final int WINDOW=10000;
  private AlarmManager mgr=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mgr=getSystemService(AlarmManager.class);
```

```
    schedule();
  }

  @Override
  public void onDestroy() {
    mgr.cancel(this);

    super.onDestroy();
  }

  @Override
  public void onAlarm() {
    Toast.makeText(this, R.string.toast, Toast.LENGTH_SHORT).show();
    Log.d(getClass().getSimpleName(), "I ran!");
    schedule();
  }

  private void schedule() {
    mgr.setWindow(AlarmManager.ELAPSED_REALTIME,
      SystemClock.elapsedRealtime()+PERIOD, WINDOW,
      getClass().getSimpleName(), this, null);
  }
}
```

(from AlarmManager/Listener/app/src/main/java/com/commonsware/android/alarm/SimpleAlarmDemoActivity.java)

`onCreate()` gets an `AlarmManager` as before. However, since this version of the app has a `minSdkVersion` of 24, we can use the version of `getSystemService()` that takes the desired system service `Class` object as a parameter and returns the system service instance in the proper data type.

Then, `onCreate()` calls `schedule()`, which in turn calls `setWindow()`. The version of `setWindow()` that we are using takes:

- the type of alarm (`ELAPSED_REALTIME`)
- the time the alarm event should fire first (`PERIOD` milliseconds from now)
- the window in which we are willing to allow Android to "flex" the actual time of the event, to perhaps save battery
- a `String` representing some tag to be used for battery usage logging purposes
- our `OnAlarmListener` implementation, which in this case happens to be the activity itself
- `null`, indicating that we want `onAlarm()` of the `OnAlarmListener` to be called on the main application thread

The alternative to `null` for the latter parameter would be a `Handler` from a `HandlerThread`, indicating that `onAlarm()` should be called on that thread.

Eventually, `onAlarm()` is called, where we show a `Toast`, log a message to Logcat,... and call `schedule()` again, so our alarm repeats.

Later, when the activity is destroyed, we call `cancel()`, passing in our `OnAlarmListener`, so all alarms tied to that listener will be discontinued.

PowerManager and WakeLocks

There are going to be times when you want the device to keep running, even though it ordinarily would go into a sleep mode, with the CPU powered down and the screen turned off. Sometimes, that will be based upon user interactions, or the lack thereof, such as keeping the screen on while playing back a video. Sometimes, that will be to allow background scheduled work to run to completion, as was introduced in the chapter on `AlarmManager`.

This chapter looks a bit more at the details of this sort of power management, including coverage of how `AlarmManager` works.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on `AlarmManager`.

Keeping the Screen On, UI-Style

If your objective is to keep the screen (and CPU) on while your activity is in the foreground, the simplest solution is to add `android:keepScreenOn="true"` to something in the activity's layout. So long as that widget or container is visible, the screen will stay on.

If you wish to do this conditionally, `setKeepScreenOn()` allows you to toggle this setting at runtime.

Once your activity is no longer in the foreground, or the widget or container is no longer visible, the effect lapses, and screen operation returns to normal.

The Role of the WakeLock

Most of the time in Android, you are developing code that will run while the user is actually using the device. Activities, for example, only really make sense when the device is fully awake and the user is tapping on the screen or keyboard.

Particularly with scheduled background tasks, though, you need to bear in mind that the device will eventually "go to sleep". In full sleep mode, the display, main CPU, and keyboard are all powered off, to maximize battery life. Only on a low-level system event, like an incoming phone call, will anything wake up the device.

Another thing that will partially wake up the phone is an `Intent` raised by the `AlarmManager`. So long as broadcast receivers are processing that `Intent`, the `AlarmManager` ensures the CPU will be running (though the screen and keyboard are still off). Once the broadcast receivers are done, the `AlarmManager` lets the device go back to sleep.

You can achieve the same effect in your code via a `WakeLock`.

One of the changes that the core Android team made to the Linux kernel was to introduce the concept of the "wakelock". In simple terms, a wakelock allows a Linux userland application — such as our Android SDK apps — to control whether or not the CPU can be powered down as part of a sleep mode. While a wakelock is in force, the CPU will remain on and processing instructions from the processes and threads that are on the device.

From the SDK, to access a wakelock, you use a `WakeLock` object, obtained from the `PowerManager` system service. When you call `acquire()` on that `WakeLock`, the CPU will remain on; when you call `release()` on that `WakeLock`, the CPU can fall back asleep, if there are no other outstanding `WakeLock`s from SDK apps or the operating system itself.

There are four types of `WakeLock` objects. All will keep the CPU on. They vary in their effects on the screen (leave it off, have it display with dim backlight, have it display with normal backlight) and any physical keys (ignore or accept). You will pass a flag into `newWakeLock()` on the `PowerManager` system service to indicate what type of `WakeLock` you want. The most common is the `PARTIAL_WAKE_LOCK`, which keeps the CPU on but leaves the screen and keyboard off — ideal for periodic background work triggered by an `AlarmManager` event.

What WakefulIntentService Does

For a `_WAKEUP` alarm, the `AlarmManager` will arrange for the device to stay awake, via a `WakeLock`, for as long as the `BroadcastReceiver`'s `onReceive()` method is executing. For some situations, that may be all that is needed. However, `onReceive()` is called on the main application thread, and Android will kill off the receiver if it takes too long.

Your natural inclination in this case is to have the `BroadcastReceiver` arrange for a `Service` to do the long-running work on a background thread, since `BroadcastReceiver` objects should not be starting their own threads. Perhaps you would use an `IntentService`, which packages up this "start a `Service` to do some work in the background" pattern. And, given the preceding section, you might try acquiring a partial `WakeLock` at the beginning of the work and release it at the end of the work, so the CPU will keep running while your `IntentService` does its thing.

This strategy will work... some of the time.

The problem is that there is a gap in `WakeLock` coverage, as depicted in the following diagram:

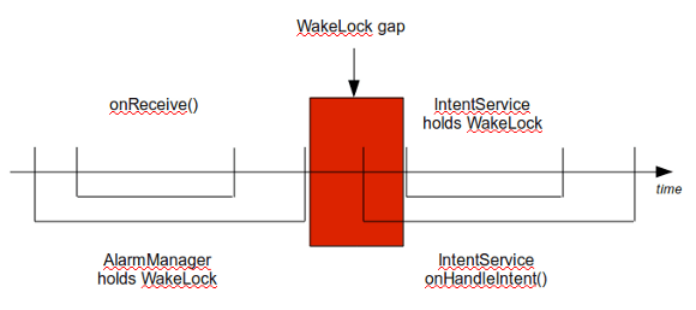


Figure 859: The WakeLock Gap

The `BroadcastReceiver` will call `startService()` to send work to the `IntentService`, but that service will not start up until after `onReceive()` ends. As a result, there is a window of time between the end of `onReceive()` and when your `IntentService` can acquire its own `WakeLock`. During that window, the device might fall back asleep. Sometimes it will, sometimes it will not.

What you need to do, instead, is arrange for overlapping `WakeLock` instances. You need to acquire a `WakeLock` in your `BroadcastReceiver`, during the `onReceive()`

execution, and hold onto that `WakeLock` until the work is completed by the `IntentService`:

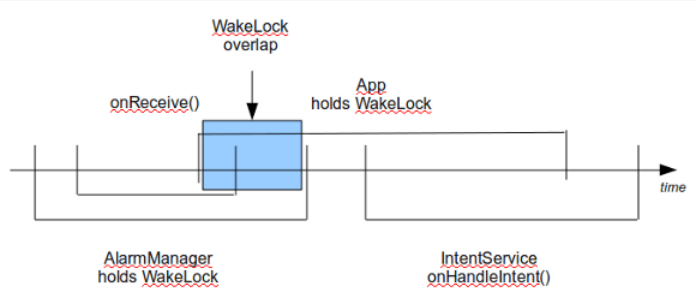


Figure 860: The WakeLock Overlap

Then you are assured that the device will stay awake as long as the work remains to be done.

The `WakefulIntentService` recipe described in its chapter does not have you manage your own `WakeLock`. That is because `WakefulIntentService` handles it for you. One reason why `WakefulIntentService` exists is to manage that `WakeLock`, because `WakeLocks` suffer from one major problem: they are not `Parcelable`, and therefore cannot be passed in an `Intent` extra. Hence, for our `BroadcastReceiver` and our `WakefulIntentService` to use the same `WakeLock`, they have to be shared via a static data member... which is icky. `WakefulIntentService` is designed to hide this icky part from you, so you do not have to worry about it.

`WakefulIntentService` also handles various edge and corner cases, such as:

- What happens if Android elects to get rid of your process due to low memory conditions?
- What happens if your `doWakefulWork()` crashes, so we do not leak the acquired `WakeLock`?
- What if your UI also sends commands to the `WakefulIntentService`, or your processing takes longer than your polling period in `AlarmManager`, so that we have more than one piece of work outstanding at a point in time?

The one requirement related to a `WakeLock` that `WakefulIntentService` imposes upon you is the `WAKE_LOCK` permission. Any code in your process that is directly manipulating `WakeLock` objects needs this permission, even if that code is from a third-party JAR like `WakefulIntentService`.

JobScheduler

`AlarmManager` was our original solution for doing work on a periodic basis. However, `AlarmManager` can readily be misused, in ways that impact the battery — this is why API Level 19 put renewed emphasis on "inexact" alarm schedules. Worse, `AlarmManager` will give us control at points in time that may be useless to us, such as giving us control when there is no Internet access, when the point of the scheduled work is to transfer some data over the Internet.

Android 5.0 introduced `JobScheduler`, which offers a more sophisticated API for handling these sorts of scenarios. This chapter will explore how to set up `JobScheduler` and use it for one-off and periodic work.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on `AlarmManager`. Also, you should have read the chapter on `PowerManager` and wakelocks.

The Limitations of AlarmManager

`AlarmManager` does its job, and frequently does it well. However, it is far from perfect:

- It does not persist its alarm schedule across reboots, forcing us to implement an `ACTION_BOOT_COMPLETED` `BroadcastReceiver` to re-establish our alarms
- It does not keep the device awake after waking it up with a `_WAKEUP` alarm, forcing us to use tools like `WakefulBroadcastReceiver` to make sure that we can get our work done without the device falling back asleep

- It gives us control even if the work we want to do is not possible, such as wanting to download material from the Internet but being woken up at points in time when we lack a working Internet connection (e.g., a WiFi-only tablet in a location for which it does not recognize any access points)
- In cases where the criteria we want cannot be met, we cannot readily implement any sort of back-off policy, except by doing the calculations ourselves and perhaps abandoning the convenient "repeating" API outright

And so on. `AlarmManager` is nice, but it would be better to have another solution.

Enter the JobScheduler

`JobScheduler` was designed to handle those four problems outlined above:

- It persists its roster of jobs and will re-establish them automatically after a reboot. Note, though, that you still have to hold the `ACTION_BOOT_COMPLETED` permission for this to work. Also note that you do not *have* to have jobs be persisted — this is an opt-in capability of `JobScheduler`.
- It handles "wakefulness" for us, via its own `WakeLock`, so we do not have to worry about it ourselves.
- It offers an API where we can specify criteria to be satisfied before we should be given control, notably a criteria indicating that we need a working network connection.
- If our criteria cannot be met, `JobScheduler` implements a configurable back-off policy, so we can slow down our attempts to get control when those attempts are regularly failing.

Employing JobScheduler

The `JobScheduler/PowerHungry` sample project demonstrates the use of `JobScheduler`, by way of comparing its use to that of `AlarmManager`.

The UI for `JobScheduler` allows you to pick from three types of event schedules: exact alarm, inexact alarm, and `JobScheduler`. You can also choose from one of four polling periods: 1 minute, 15 minutes, 30 minutes, and 60 minutes:

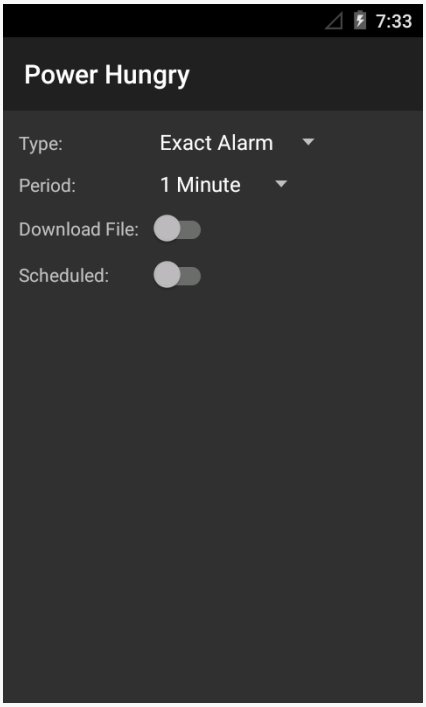


Figure 861: PowerHungry Demo, As Initially Launched

A `Switch` allows you to determine whether you are simply getting control at those points in time to just log to Logcat, or whether you are going to try to do some work at those points in time. Specifically, the "work" is to download a file, using `HttpUrlConnection`.

The bottom `Switch` toggles on and off the event schedules. When the event schedules are toggled on, you cannot manipulate the rest of the UI — you need to turn off the events in order to change the event configuration.

Note that none of this information is persisted. This is a lightweight demo; it is expected that you are keeping this UI in the foreground while a test is running.

Defining and Scheduling the Job

The "job" is defined as an instance of `JobInfo`, typically created using an instance of `JobInfo.Builder` to configure a `JobInfo` using a fluent builder-style API. We teach the `JobInfo` the work to do and when to do it, then use a `JobScheduler` to actually schedule the job.

In the sample app, this work is mostly accomplished via a `manageJobScheduler()` method on the `MainActivity` class:

```
private void manageJobScheduler(boolean start) {
  if (start) {
    JobInfo.Builder b=new JobInfo.Builder(JOB_ID,
        new ComponentName(this, DemoJobService.class));
    PersistableBundle pb=new PersistableBundle();

    if (download.isChecked()) {
      pb.putBoolean(KEY_DOWNLOAD, true);
      b.setExtras(pb).setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);
    } else {
      b.setRequiredNetworkType(JobInfo.NETWORK_TYPE_NONE);
    }

    b.setPeriodic(getPeriod()).setPersisted(false)
        .setRequiresCharging(false).setRequiresDeviceIdle(true);

    jobs.schedule(b.build());
  }
  else {
    jobs.cancel(JOB_ID);
  }
}
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

The `start` parameter to `manageJobScheduler()` is driven by the bottom `Switch` widget. A `start` value of `true` means that we should start up the job; a value of `false` means that we should cancel any existing job.

If `start` is `true`, we begin by creating a `JobInfo.Builder`, supplying two key pieces of data:

- an `int` that will serve as the job ID, which needs to be unique to our app but does not have to be unique for the whole device
- a `ComponentName` identifying the `JobService` that will actually implement the work of the job itself

The primary way of passing data from the scheduling code (our activity) and the job-implementing code (`JobService`) is by means of a `PersistableBundle` – a `Bundle`-like object that can be persisted to disk. `PersistableBundle` was introduced in API Level 21, but at that time it inexplicably lacked support for `boolean` values. API Level 22 added `getBoolean()` and `putBoolean()` to `PersistableBundle`, and this

sample project has `minSdkVersion` of 22 to be able to take advantage of it. If you wanted to use this sample on API Level 21, you would need to convert the `boolean` into something else, such as `0` and `1` `int` values.

Our `PersistableBundle` can have more data than just this one extra, though that is all we need in this case. We attach the `PersistableBundle` to the `JobInfo` via the `setExtras()` method on the `JobInfo.Builder`.

We can also call methods on the `JobInfo.Builder` to configure the criteria that should be satisfied before giving us control. In our case, one criterion that we need is to have a network connection, but only if we are supposed to be downloading a file. So, we call `setRequiredNetworkType()` in either case, indicating that we either want `ANY` type of network connection (metered or unmetered) or `NONE`.

Other criteria-defining methods that we invoke include `setRequiresCharging()` (set to `false` to indicate we want control even if we are on battery) and `setRequiresDeviceIdle()` (set to `true` to indicate that we want control only if the user is not using it).

In the case of this sample, we want to do this work every so often, based upon the period chosen by the user in the bottom `Spinner` and retrieved via the `getPeriod()` method. So, we call `setPeriodic()` on the `JobInfo.Builder` to request getting control with that frequency, bearing in mind that this is merely a hint, not a requirement, and we may get control more or less frequently than this.

We also call `setPersisted(false)` to indicate that we do not need for this job to be persisted, so it will be lost on a reboot. If we instead called `setPersisted(true)`, the manifest would need to request the `RECEIVE_BOOT_COMPLETED` permission to have the job be re-created at boot time.

Finally, we call `schedule()` on a `JobScheduler` instance named `jobs` to schedule the job.

The `jobs` data member is populated up in `onReady()` of the activity, where `onReady()` is a callback from the `AbstractPermissionActivity`:

```
    jobs=(JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

`AbstractPermissionActivity`, seen elsewhere in this book, handles requesting the runtime permission for external storage that we are going to need later on.

If the `start` parameter to `manageJobScheduler()` is `false`, we call `cancel()` on the `JobScheduler`, passing in our unique job ID (`JOB_ID`) to indicate what job to cancel. Or, we could have called `cancelAll()`, which would cancel all jobs scheduled by our application.

Implementing the Job

The work for the job itself is handled by a `JobService`. This is a subclass of `Service` that we, in turn, extend ourselves, overriding two job-specific callback methods to actually do the work: `onStartJob()` and `onStopJob()`.

The `JobService` in our sample app is `DemoJobService`:

```
package com.commonsware.android.job;

import android.app.job.JobParameters;
import android.app.job.JobService;
import android.os.PersistableBundle;
import android.util.Log;

public class DemoJobService extends JobService {
  private volatile Thread job=null;

  @Override
  public boolean onStartJob(JobParameters params) {
    PersistableBundle pb=params.getExtras();

    if (pb.getBoolean(MainActivity.KEY_DOWNLOAD, false)) {
      job=new DownloadThread(params);
      job.start();

      return(true);
    }

    Log.d(getClass().getSimpleName(), "job invoked");

    return(false);
  }

  @Override
  synchronized public boolean onStopJob(JobParameters params) {
    if (job!=null) {
      Log.d(getClass().getSimpleName(), "job interrupted");
      job.interrupt();
    }
```

```
    return(false);
  }

  synchronized private void clearJob() {
    job=null;
  }

  private class DownloadThread extends Thread {
    private final JobParameters params;

    DownloadThread(JobParameters params) {
      this.params=params;
    }

    @Override
    public void run() {
      Log.d(getClass().getSimpleName(), "job begins");
      new DownloadJob().run();
      Log.d(getClass().getSimpleName(), "job ends");
      clearJob();
      jobFinished(params, false);
    }
  }
}
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/DemoJobService.java)

`onStartJob()` is passed a `JobParameters`. This serves both as a "handle" identifying a particular job invocation and giving us access to the job ID (`getJobId()`) and `PersistableBundle` of extras (`getExtras()`) that were set up by our `JobInfo` when we scheduled the job.

`onStartJob()` needs to return `true` if we have successfully forked a background thread to do the work, or `false` if no work needs to be done. In our case, this is determined by whether or not we want to try to download a file. In a production-grade app, this may be determined by whether there is any work to be done (e.g., "do we have entries in the upload queue?").

In `onStartJob()`, we check the `PersistableBundle` to see if we are supposed to download a file. If we are, we fork a `DownloadThread` to do that work, then return `true`. Otherwise, we return `false`.

Because this sample app illustrates the difference in behavior between `JobScheduler` and `AlarmService`, we want to isolate the actual download-the-file logic into a

common implementation that can be used from either code path. That takes the form of a `DownloadJob`, which implements `Runnable` and does the download work when it is `run()`:

```
package com.commonsware.android.job;

import android.net.Uri;
import android.os.Environment;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

class DownloadJob implements Runnable {
  static final Uri TO_DOWNLOAD=
      Uri.parse("https://commonsware.com/Android/excerpt.pdf");

  @Override
  public void run() {
    try {
      File root=Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

      root.mkdirs();

      File output=new File(root, TO_DOWNLOAD.getLastPathSegment());

      if (output.exists()) {
        output.delete();
      }

      URL url=new URL(TO_DOWNLOAD.toString());
      HttpURLConnection c=(HttpURLConnection)url.openConnection();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;

        while ((len=in.read(buffer)) >= 0) {
          out.write(buffer, 0, len);
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
        c.disconnect();
      }
    }
    catch (IOException e2) {
```

```
      Log.e("DownloadJob", "Exception in download", e2);
    }
  }
}
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/DownloadJob.java)

`DownloadThread` delegates to `DownloadJob` to do the actual work. However, when the work is complete, it then calls `jobFinished()` on the `DemoJobService`. `jobFinished()`, as the name suggests, tells the framework that we are finished doing the work associated with this job. If the job succeeded, we pass `false` as the second parameter, to indicate that this job does not need to be rescheduled. If, on the other hand, we were unable to actually do the work (e.g., we cannot connect to the desired server, perhaps due to server maintenance), we would pass `true` as the second parameter, to request that this job be rescheduled to be invoked again shortly, so that we can retry the operation.

Our `onStopJob()` method will be called by Android if environmental conditions have changed and we should stop the background work that we are doing. For example, we asked to do this work when the device was idle — if the user picks up the device and starts using it, we should stop our background work. In this case, if the job thread is still outstanding, we `interrupt()` it. `onStopJob()` should return `true` if this job is still needed and should be retried, or `false` otherwise. Most short-period periodic jobs should return `false`, to just worry about the next job in the next period, and that is what `onStopJob()` does here. One-time jobs, or jobs with long periods (e.g., a day), may wish to return `true` to ensure that they will get another chance to do the desired work. We will cover more about this issue later in this chapter.

Wiring in the Job Service

Since a `JobService` is a `Service`, we need the corresponding `<service>` element in the manifest. For a `JobService`, the `<service>` element is perfectly normal... with one exception:

```
    <service
      android:name=".DemoJobService"
      android:permission="android.permission.BIND_JOB_SERVICE" />
```

(from JobScheduler/PowerHungry/app/src/main/AndroidManifest.xml)

You need to defend the service with the `BIND_JOB_SERVICE` permission. This only allows code that holds the `BIND_JOB_SERVICE` permission to start or bind to this service, which should limit it to the OS itself.

The Rest of the Sample

As noted earlier, the UI for our activity is a pair of `Spinner` widgets, along with a pair of `Switch` widgets:

```
<?xml version="1.0" encoding="utf-8"?>

<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp"
    android:useDefaultMargins="true">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/type_label"
        android:layout_row="0"
        android:layout_column="0"/>

    <Spinner
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/type"
        android:layout_row="0"
        android:layout_column="1"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/period_label"
        android:layout_row="1"
        android:layout_column="0"/>

    <Spinner
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/period"
        android:layout_row="1"
        android:layout_column="1"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/download_label"
        android:layout_row="2"
```

```
        android:layout_column="0" />

    <Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/download"
        android:layout_row="2"
        android:layout_column="1" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/scheduled_label"
        android:layout_row="3"
        android:layout_column="0" />

    <Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/scheduled"
        android:layout_row="3"
        android:layout_column="1" />
</GridLayout>
```

(from JobScheduler/PowerHungry/app/src/main/res/layout/main.xml)

`onReady()` of `MainActivity` sets up the UI, including populating the two `Spinner` widgets based on `<string-array>` resources and hooking up the activity to respond to changes in the checked state of the `scheduled` `Switch` widget:

```
@SuppressWarnings("ResourceType")
@Override
public void onReady(Bundle savedInstanceState) {
  setContentView(R.layout.main);
  type=findViewById(R.id.type);

  ArrayAdapter<String> types=
      new ArrayAdapter<String>(this,
          android.R.layout.simple_spinner_item,
          getResources().getStringArray(R.array.types));

  types.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
  type.setAdapter(types);

  period=findViewById(R.id.period);

  ArrayAdapter<String> periods=
      new ArrayAdapter<String>(this,
          android.R.layout.simple_spinner_item,
          getResources().getStringArray(R.array.periods));

  periods.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
```

```
    period.setAdapter(periods);

    download=findViewById(R.id.download);
    scheduled=findViewById(R.id.scheduled);
    scheduled.setOnCheckedChangeListener(this);

    alarms=(AlarmManager)getSystemService(ALARM_SERVICE);
    jobs=(JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
  }
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

When the user toggles the `scheduled` `Switch` widget, we examine the `type` `Spinner` and route control to a method dedicated for handling that particular type of periodic request, such as the `manageJobScheduler()` method we saw earlier in this chapter:

```
  private void manageJobScheduler(boolean start) {
    if (start) {
      JobInfo.Builder b=new JobInfo.Builder(JOB_ID,
          new ComponentName(this, DemoJobService.class));
      PersistableBundle pb=new PersistableBundle();

      if (download.isChecked()) {
        pb.putBoolean(KEY_DOWNLOAD, true);
        b.setExtras(pb).setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);
      } else {
        b.setRequiredNetworkType(JobInfo.NETWORK_TYPE_NONE);
      }

      b.setPeriodic(getPeriod()).setPersisted(false)
          .setRequiresCharging(false).setRequiresDeviceIdle(true);

      jobs.schedule(b.build());
    }
    else {
      jobs.cancel(JOB_ID);
    }
  }
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

Our `onCheckedChanged()` for the `schedule` `Switch` also calls a `toggleWidgets()` method that enables or disables the other widgets, depending upon whether the `schedule` `Switch` is checked or unchecked:

```
  private void toggleWidgets(boolean enable) {
    type.setEnabled(enable);
    period.setEnabled(enable);
```

```
    download.setEnabled(enable);
  }

  private void manageExact(boolean start) {
    if (start) {
      long period=getPeriod();

      PollReceiver.scheduleExactAlarm(this, alarms, period,
          download.isChecked());
    }
    else {
      PollReceiver.cancelAlarm(this, alarms);
    }
  }

  private void manageInexact(boolean start) {
    if (start) {
      long period=getPeriod();

      PollReceiver.scheduleInexactAlarm(this, alarms, period,
          download.isChecked());
    }
    else {
      PollReceiver.cancelAlarm(this, alarms);
    }
  }

  private void manageJobScheduler(boolean start) {
    if (start) {
      JobInfo.Builder b=new JobInfo.Builder(JOB_ID,
          new ComponentName(this, DemoJobService.class));
      PersistableBundle pb=new PersistableBundle();

      if (download.isChecked()) {
        pb.putBoolean(KEY_DOWNLOAD, true);
        b.setExtras(pb).setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);
      } else {
        b.setRequiredNetworkType(JobInfo.NETWORK_TYPE_NONE);
      }

      b.setPeriodic(getPeriod()).setPersisted(false)
          .setRequiresCharging(false).setRequiresDeviceIdle(true);

      jobs.schedule(b.build());
    }
    else {
      jobs.cancel(JOB_ID);
    }
```

```
  }

  private long getPeriod() {
    return(PERIODS[period.getSelectedItemPosition()]);
  }
}
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

If the user had chosen an exact alarm, `onCheckedChanged()` routes control to `manageExact()`:

```
  private void manageExact(boolean start) {
    if (start) {
      long period=getPeriod();

      PollReceiver.scheduleExactAlarm(this, alarms, period,
          download.isChecked());
    }
    else {
      PollReceiver.cancelAlarm(this, alarms);
    }
  }
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

It, in turn, routes control over to a `PollReceiver`, a `BroadcastReceiver` that is set up for handling our alarms:

```
package com.commonsware.android.job;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class PollReceiver extends BroadcastReceiver {
  static final String EXTRA_PERIOD="period";
  static final String EXTRA_IS_DOWNLOAD="isDownload";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    boolean isDownload=i.getBooleanExtra(EXTRA_IS_DOWNLOAD, false);

    DemoScheduledService.enqueueWork(ctxt,
```

```
        new Intent(ctxt, DemoScheduledService.class)
            .putExtra(EXTRA_IS_DOWNLOAD, isDownload));

    long period=i.getLongExtra(EXTRA_PERIOD, -1);

    if (period>0) {
      scheduleExactAlarm(ctxt,
          (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE),
          period, isDownload);
    }
  }

  static void scheduleExactAlarm(Context ctxt, AlarmManager alarms,
                                 long period, boolean isDownload) {
    Intent i=new Intent(ctxt, PollReceiver.class)
        .putExtra(EXTRA_PERIOD, period)
        .putExtra(EXTRA_IS_DOWNLOAD, isDownload);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    alarms.setExact(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()+period, pi);
  }

  static void scheduleInexactAlarm(Context ctxt, AlarmManager alarms,
                                   long period, boolean isDownload) {
    Intent i=new Intent(ctxt, PollReceiver.class)
        .putExtra(EXTRA_IS_DOWNLOAD, isDownload);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    alarms.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()+period, period, pi);
  }

  static void cancelAlarm(Context ctxt, AlarmManager alarms) {
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    alarms.cancel(pi);
  }
}
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/PollReceiver.java)

This sample app has a `targetSdkVersion` of 21. Hence, on Android 5.0 devices — the ones that have `JobScheduler`, we cannot set up exact repeating alarms. Our only option is to handle the repeating work ourselves.

Hence, `scheduleExactAlarm()` creates a broadcast `PendingIntent`, on an `Intent` pointing at our `PollReceiver`, with a pair of extras indicating the polling period and whether or not we should be downloading a file. It then uses `setExact()` on an `AlarmManager` to schedule a *one-off* event to occur one polling period from now.

That, in turn, will trigger `onReceive()` of the `PollReceiver`. Here, we call `startWakefulService()` to have our work be done by a `DemoScheduledService`. In addition, if we have a polling period, that means that this is an exact alarm, and we call `scheduleExactAlarm()` to set up the next occurrence of this "repeating" event.

`DemoScheduledService` is simply a `JobIntentService` wrapper around the `DownloadJob` that we used with `DemoJobService`.

`cancelAlarm()` on `PollReceiver` — called by `manageExact()` when we are stopping the repeating event — creates an equivalent `PendingIntent` to the ones used for the `AlarmManager` events, and uses that with `cancel()` on `AlarmManager` to cancel those events.

If the user had chosen an inexact alarm, `onCheckedChanged()` routes control to `manageInexact()`:

```
private void manageInexact(boolean start) {
  if (start) {
    long period=getPeriod();

    PollReceiver.scheduleInexactAlarm(this, alarms, period,
        download.isChecked());
  }
  else {
    PollReceiver.cancelAlarm(this, alarms);
  }
}
```

(from JobScheduler/PowerHungry/app/src/main/java/com/commonsware/android/job/MainActivity.java)

It uses the same recipe as `manageExact()`, except that it calls `scheduleInexactAlarm()` on `PollReceiver`. `scheduleInexactAlarm()`, in turn, uses `setInexactRepeating()` on `AlarmManager` to arrange to get control every so often.

Pondering Backoff Criteria

Sometimes, even with the Internet-availability checks offered by `JobScheduler`, you find that you cannot actually do the job you scheduled. Perhaps the server is down

for maintenance, or has been replaced by a honeycomb frame, or something. In this case, while you failed to do the job now, you may want to try again later.

Sometimes, "later" can just be handled by your existing `JobScheduler` setup. If the job in question is a periodic job, and missing a whole period is not a big problem, you might just continue on normally.

However, sometimes you will want the job to be retried, either because:

- it was a one-shot job, not a periodic one, or
- the period of the job is fairly long (e.g., once per day) and you want to retry well before the job is scheduled to happen again

Requesting that a job be retried is handled by the `boolean` parameter to `jobFinished()` or the `boolean` return value from `onStopJob()`. `true` means that you want the job to be rescheduled; `false` means that it is OK to skip the job entirely.

Given that you use `true` for either `jobFinished()` or `onStopJob()`, there are three possible options for how to request and retry a failed job:

- What happens for a job which you requested only run when the device is idle
- What happens for other jobs by default
- How you can influence the timing of when the job is retried, known as the "backoff criteria"

Idle Jobs

If you requested idle-only jobs, if the user wakes up the device while the job is going on, you will be called with `onStopJob()`. Ideally, you then stop the background work and return `true` or `false` from `onStopJob()` to determine if the job should be rescheduled.

If you request a job be rescheduled, when that job is set up to only run when the device is idle, the job is simply "put back in the queue" to be tried again during the next idle window.

Default Behavior

If, for a non-idle-only job, you use `true` for `jobFinished()` or `onStopJob()`, the next time to try will be calculated using the default backoff criteria, which has a time of 30 seconds and a policy of `BACKOFF_POLICY_EXPONENTIAL`.

What this means is that the first time you use `true`, your job will be tried again 30 seconds later. If you use `true` again for that job, it will be tried again 60 seconds later. If you use `true` again, it would be tried 120 seconds later — in other words, each job failure will reschedule using the formula $2^{n-1}t$, where `n` is the number of failures and `t` is 30 seconds.

However, there is a cap of 18,000,000 milliseconds, or what normal people would refer to as "5 hours". That is the most your job will be delayed, regardless of how many failures you have.

Custom Backoff Criteria

You can change the backoff criteria for non-idle-only jobs via a call to `setBackoffCriteria()` on your `JobInfo.Builder`, where you provide your own time (measured in milliseconds) and policy (`BACKOFF_POLICY_EXPONENTIAL` or `BACKOFF_POLICY_LINEAR`).

As noted above, the formula for exponential backoff rescheduling is $2^{n-1}t$, where `n` is the number of failures and `t` is your chosen time.

The formula for linear backoff rescheduling is `n*t`, where `n` is the number of failures and `t` is your chosen time.

Other JobScheduler Features

There are a few other options for scheduling jobs that may be of use to you in select circumstances:

- `JobInfo.Builder` has `setOverrideDeadline()`, which indicates a maximum delay for this job before it will be executed even if other criteria (e.g., idleness) have not been met. Note that this is only available on one-shot jobs, not periodic jobs.
- The `JobParameters` passed to `onStartJob()` has an `isOverrideDeadlineExpired()` method. This will return `true` if the job was

executed early due to a `setOverrideDeadline()` value being met. This will indicate to you that your requirements may not be met (e.g., Internet access) and you will need to double-check those things yourself.
- `JobInfo.Builder` has `setMinimumLatency()` which sets a minimum delay time; the job will not be considered until at least this amount of time has elapsed. Note that this is only available on one-shot jobs, not periodic jobs.

Also, `JobScheduler` has a `getAllPendingJobs()` method, that returns a `List` of `JobInfo` objects representing "the jobs registered by this package that have not yet been executed". Presumably, this includes the next occurrence of any periodic jobs and any jobs that are blocked pending a backoff delay, though the documentation is unclear on this point.

JobScheduler Period Limits

As of Android 7.0, `JobScheduler` does not support jobs running more frequently than once every 15 minutes.

GcmNetworkManager

As noted earlier in this chapter, Android 5.0 added `JobScheduler`. However, Google did not release any sort of backport of this, as that would be difficult to do on a whole-device basis. They did not even implement a `JobSchedulerCompat`, hampering adoption.

Firebase now has a `GcmNetworkManager` that, despite the name, is basically a backport of `JobScheduler`. In fact, it will delegate to `JobScheduler` on Android 5.0+ devices. It is unclear how old of an Android OS version `GcmNetworkManager` supports, but it is likely to work on more devices than does `JobScheduler`.

However, it does introduce a tie to Google Play Services, which will not be appropriate for all apps.

Periodic Work, Across Device Versions

Of course, all of this is a pain. And, where there is pain, somebody eventually creates a library to try to ease that pain.

Evernote — the NSaaS (note storage as a service) provider — has released `android-job`, a library that offers a single API that uses `GcmNetworkManager` (if you opt into it), `JobScheduler` (on API Level 21+, for inexact jobs), or `AlarmManager` (for API Level 19 and below, plus for exact jobs). This library can simplify your code, by handling the version-specific logic for you.

The `JobScheduler/Dispatcher` sample project demonstrates the use of `android-job`. It is based upon the `PowerHungry` sample app, adding in a new option for using this new library.

The Dependency

Evernote publishes `android-job` as an artifact, so adding it to your project is as simple as a single line in your `build.gradle` file:

```
dependencies {
    implementation 'com.android.support:support-v13:27.0.2'
    implementation 'com.evernote:android-job:1.2.1'
}
```

(from JobScheduler/Dispatcher/app/build.gradle)

The Job

With `AlarmManager`, usually your periodic work is handled by a combination of a `WakefulBroadcastReceiver` and an `IntentService`. With `JobScheduler`, your work is handled by a `JobService`. With `android-job`, your work is handled by custom subclasses of a library-supplied `Job` class.

Your `Job` subclass needs to override `onRunJob()`. Akin to `onStartJob()` of a `JobService`, you get a `Params` object that you can use to identify the details of this specific job. Based on the library's implementation and JavaDocs, you should do the work for your job directly in `onRunJob()`, returning one of three values:

- `Result.SUCCESS`, meaning that life is good
- `Result.RESCHEDULE`, meaning that you did not do the work, and it should be rescheduled to be tried again shortly
- `Result.FAILURE`, meaning that you did not do the work, but there is no reason to reschedule the job

The sample app has a `DemoUnifiedJob` that handles all of this:

```
package com.commonsware.android.job;

import android.support.annotation.NonNull;
import android.util.Log;
import com.evernote.android.job.Job;

public class DemoUnifiedJob extends Job {
  static final String JOB_TAG=
    DemoUnifiedJob.class.getCanonicalName();

  @NonNull
  @Override
  protected Result onRunJob(Params params) {
    Log.d(getClass().getSimpleName(), "scheduled unified work begins");

    if (getParams()
          .getExtras()
          .getBoolean(PollReceiver.EXTRA_IS_DOWNLOAD, false)) {
      new DownloadJob().run();  // do synchronously, as we are on
                                // a background thread already
    }

    Log.d(getClass().getSimpleName(), "scheduled unified work ends");

    return(Result.SUCCESS);
  }
}
```

(from JobScheduler/Dispatcher/app/src/main/java/com/commonsware/android/job/DemoUnifiedJob.java)

As will be seen later when we schedule this work, we add in our `EXTRA_IS_DOWNLOAD` `boolean` value, akin to how we handled this with `JobScheduler`, to know whether or not we are supposed to download the file or not.

The JobCreator

While executing jobs is similar to `JobScheduler` (just simpler), actually setting up jobs is quite a bit more cumbersome.

The first step towards setting up jobs is to create a `JobCreator`. This class is simply a way to tie a `String` to a `Job` subclass. In the `create()` method, given a `String`, you return an instance of the associated `Job` subclass, as is done in the sample app's `DemoUnifiedJobCreator`:

```
package com.commonsware.android.job;
```

```
import com.evernote.android.job.Job;
import com.evernote.android.job.JobCreator;

public class DemoUnifiedJobCreator implements JobCreator {
  @Override
  public Job create(String tag) {
    if (DemoUnifiedJob.JOB_TAG.equals(tag)) {
        return(new DemoUnifiedJob());
    }

    throw new IllegalArgumentException("Job tag not recognized: "+tag);
  }
}
```

(from JobScheduler/Dispatcher/app/src/main/java/com/commonsware/android/job/DemoUnifiedJobCreator.java)

The Application

The recommended pattern for setting up `android-job` is to use a custom `Application` subclass. It needs to set up a `JobManager` singleton and register any `JobCreator` classes that you might have.

The sample app does this in `DemoUnifiedApplication`:

```
package com.commonsware.android.job;

import android.app.Application;
import com.evernote.android.job.JobManager;

public class DemoUnifiedApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    JobManager
      .create(this)
      .addJobCreator(new DemoUnifiedJobCreator());
  }
}
```

(from JobScheduler/Dispatcher/app/src/main/java/com/commonsware/android/job/DemoUnifiedApplication.java)

That, in turn, is set up as our app's `Application` subclass via the `android:name` attribute on the `<application>` element in the manifest:

```
<application
  android:name=".DemoUnifiedApplication"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name">
```

(from JobScheduler/Dispatcher/app/src/main/AndroidManifest.xml)

Scheduling Jobs

Given all of that prep work, we can now actually schedule jobs to be executed. The flow is very similar to that of `JobScheduler`:

- Create and populate a `JobRequest.Builder` with the details of the job, notably the `String` to identify which `Job` subclass to use (by way of the `JobCreator`)
- `build()` the `JobRequest` using that `Builder`, and call `schedule()` on it to schedule the job
- Hold onto the `int` returned by `schedule()` and use that to `cancel()` the job later on, if needed

```
private void manageUnified(boolean start) {
  if (start) {
    final JobRequest.Builder b=
      new JobRequest.Builder(DemoUnifiedJob.JOB_TAG);
    PersistableBundleCompat extras=new PersistableBundleCompat();

    if (download.isChecked()) {
      extras.putBoolean(KEY_DOWNLOAD, true);
      b
        .setExtras(extras)
        .setRequiredNetworkType(JobRequest.NetworkType.CONNECTED);
    }
    else {
      b.setRequiredNetworkType(JobRequest.NetworkType.ANY);
    }

    b
      .setPeriodic(getPeriod())
      .setRequiresCharging(false)
      .setRequiresDeviceIdle(true);

    unifiedJobId=b.build().schedule();
  }
  else {
    JobManager.instance().cancel(unifiedJobId);
```

```
    }
  }
```

(from JobScheduler/Dispatcher/app/src/main/java/com/commonsware/android/job/MainActivity.java)

Enabling GcmNetworkManager Support

By default, `android-job` will use `JobScheduler` and/or `AlarmManager`. However, if you add the `play-services-gcm` dependency to your project (version 9.4.0 or higher), and configure a specific `<service>` element in your manifest, then `android-job` will also consider using `GcmNetworkManager`. Details for this are available in the project documentation.

Android 6.0 and "the War on Background Processing"

Google has been increasingly aggressive about trying to prevent background work, particularly while the device is deemed to be idle, in an effort to improve battery life. In Android 4.4 (API Level 19), we were given a strong "nudge" to use inexact alarms. In Android 5.0 (API Level 21), we were given `JobScheduler` as a smarter `AlarmManager`, but one that also emphasizes inexact schedules.

In Android 6.0, Google broke out more serious weaponry in the war against background work, in ways that are going to cause a fair bit of pain and confusion for users.

Doze Mode

If the device's screen is off, the device is not being charged, and the device does not appear to be moving (as determined via sensors, like the accelerometer), an Android 6.0+ device will go into "Doze mode". This mode is reminiscent of similar modes used by specific device manufacturers, such as SONY's STAMINA mode.

While in "Doze mode", your scheduled alarms (with `AlarmManager`), jobs (with `JobScheduler`), and syncs (with `SyncManager`) will be ignored by default, except during occasional "idle maintenance windows". In short, much of what your user thinks will happen in the background will not happen.

App Standby Mode

Further compounding the problem from “Doze mode” is “app standby”.

After some undefined period of time, an app that has not been in the foreground (or is showing a `Notification`) will be put into “standby” state. While the app is in “standby”:

- If the device is unplugged, the app behaves as though the device is in “Doze mode”, with background access degrading over time to a point where the app will only get network access in the background around once per day
- If the device is plugged in, the app behaves normally

How to Win the War

The vision behind “the war on background processing” is to improve battery life, particularly while the device is not being used (Doze mode) or for apps that are not being used (app standby). However, any number of apps will have their behavior severely compromised by these changes.

Here are some techniques for helping your app behave better on Android 6.0+.

GCM

If you are using Google Cloud Messaging (GCM), and you send a “high-priority tickle” to the app on a device, that may allow you to run then, despite being in Doze mode or app standby mode. However, this implies that you have all the plumbing set up for GCM, that the device has an active network connection, etc. Also, this requires you to adopt GCM, which has its issues (no service-level agreement, Google has access to all of the messages, etc.).

…AndAllowWhileIdle()

`AlarmManager` now has two additional methods:

- `setAndAllowWhileIdle()`
- `setExactAndAllowWhileIdle()`

These work better in Doze mode and app standby mode, allowing you to get control briefly even if otherwise you would not. However:

- While in those modes, these alarms will occur at most once every 15 minutes, except during the aforementioned "idle maintenance windows"
- There is no guarantee of how long you will be able to keep the device awake
- There is no guarantee that you can access the Internet

Use a Foreground Service

While not officially documented, Dianne Hackborn (a core Android developer) wrote in a comment on a Google+ post:

> Apps that have been running foreground services (with the associated notification) are not restricted by doze.

The Whitelist

Users have the ability to disable these "battery optimizations" for an individual app, allowing it to run closer to normally. On the "Apps" screen in Settings, there is now a gear icon in the action bar:

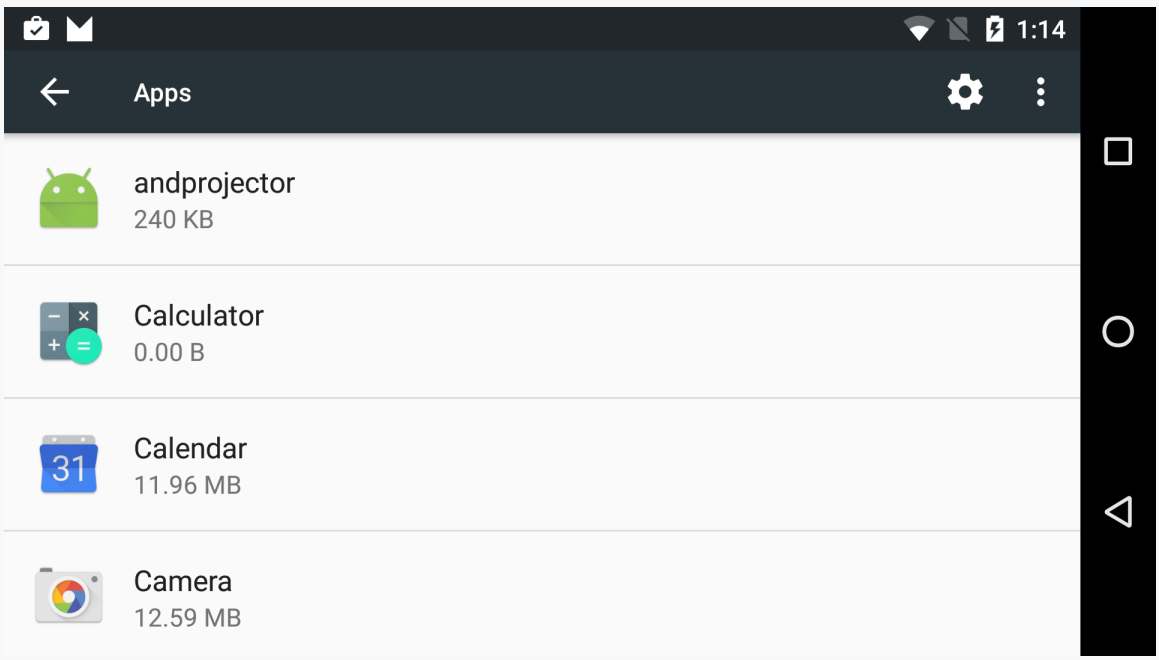


Figure 862: Android 6.0, Settings App, Apps Screen

Tapping that brings up a “Configure apps” screen. On there is a “Battery optimization” entry. Tapping on that will initially show the apps for which battery optimizations will be ignored (a.k.a., “Not optimized”):

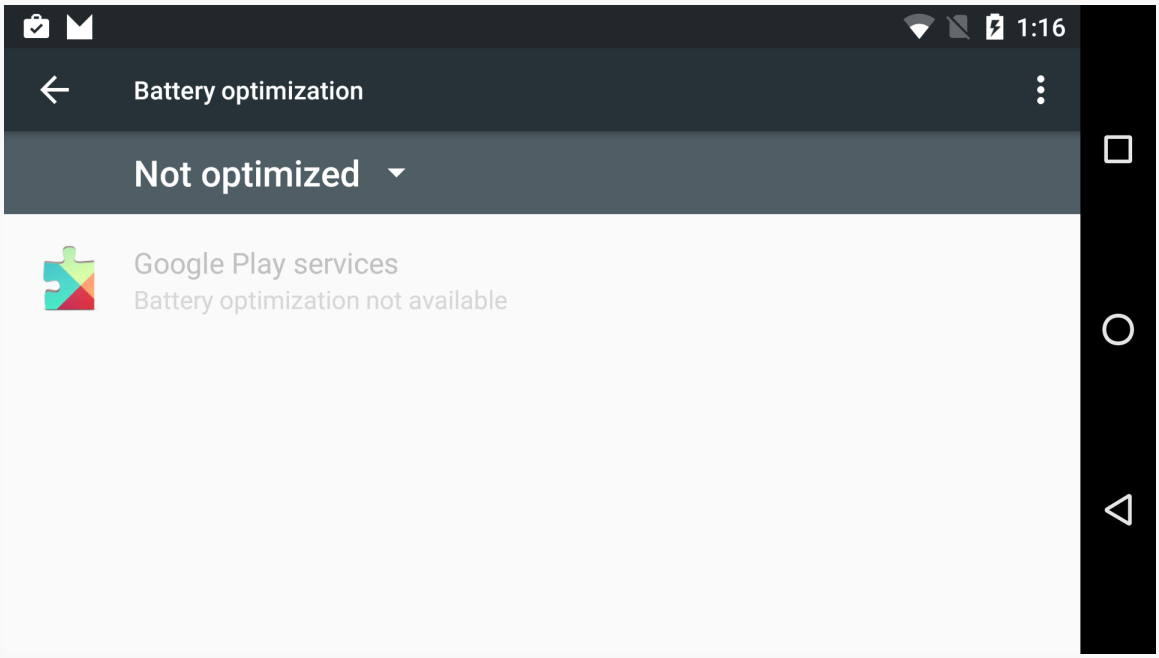


Figure 863: Android 6.0, Settings App, Battery Optimization Screen

If the user toggles the “Not optimized” drop-down to “All apps” and taps on one of those apps, the user can elect to decide whether to “optimize” the app (and cause app standby to trigger) or not:

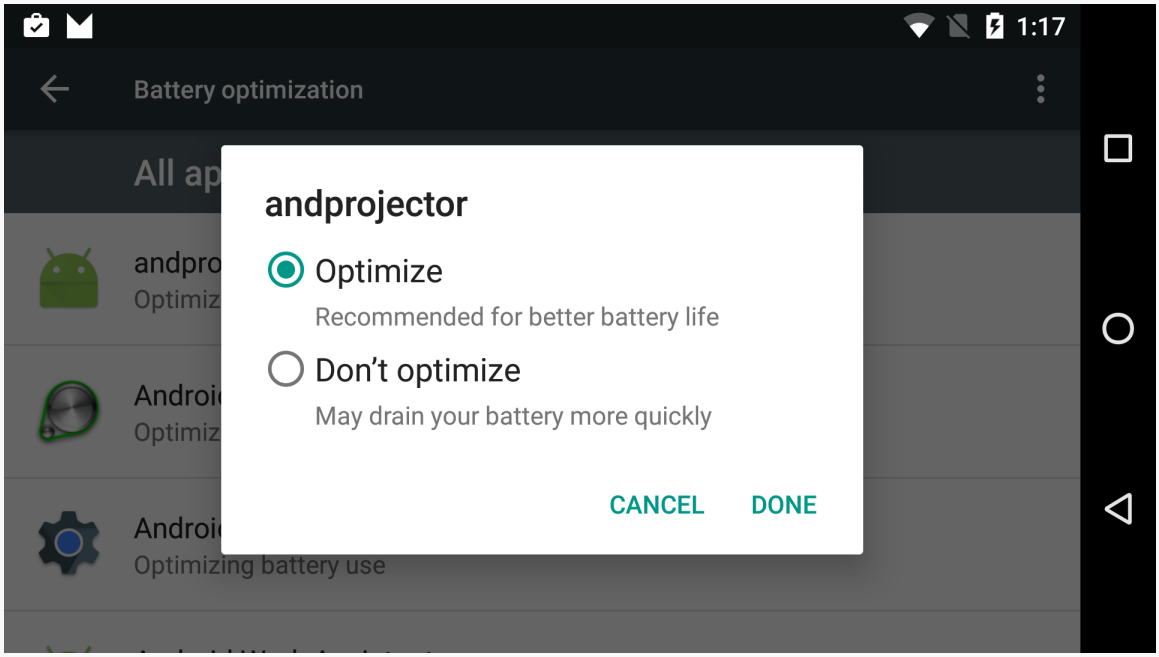


Figure 864: Android 6.0, Settings App, Battery Optimization Options Dialog

This "whitelist" of apps allows you to hold wakelocks and access the network. It does *not* change the behavior of `AlarmManager`, `JobScheduler`, or `SyncManager` — those things will still fire far less frequently in Doze mode or in app standby.

To determine if your app is already on the whitelist, you can call `isIgnoringBatteryOptimizations()` on a `PowerManager` instance.

If you would like to lead the user over to the screen where they can generally configure the whitelist, use an `ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS` `Intent` with `startActivity()`:

```
startActivity(new Intent(Settings.ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS));
```

If you would like to drive the user straight to the screen where they can add your specific app to the whitelist:

- Request the `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` permission via a `<uses-permission>` element in the manifest
- Create a `package:` `Uri` pointing to your app
- Wrap that `Uri` in an `ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` `Intent`

- Call `startActivity()` with that `Intent`

```
Intent i=new Intent(Settings.ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS,
                    Uri.parse("package:" + getPackageName()));

startActivity(intent);
```

Note, though, that using this may cause your app to be banned on the Play Store, even though it is a legitimate part of the Android SDK.

While the whitelist existed in the first developer preview of Android 6.0, its role was expanded very late in the process, as originally it did not affect Doze mode. The rationale appears to be for apps that cannot use GCM as the trigger mechanism to do background work, particularly if they need something else network-based as the trigger. For example, SIP clients, XMPP clients, MQTT clients, and so on are idle until a message comes in on an open network connection, yet none of those can be readily converted to use GCM. The whitelist allows apps to behave as they did prior to Android 6.0, though it requires user involvement.

However, any app can use this whitelist approach to return to more-normal behavior. The biggest limitation is for apps that relied upon `AlarmManager`, `JobScheduler`, or `SyncAdapter` as their triggers, as those are still crippled, regardless of whitelist status. The best you can get is ~15 minute periods, via `setExactAndAllowWhileIdle()`.

If you are sure that you need polling more frequently than that, and you are sure that the user will value that polling, your primary option is to use a foreground `Service` (or whitelisted app) and Java's `ScheduledExecutorService` to get control every so often, using a partial wakelock to keep the CPU powered on *all the time*. From a battery standpoint, this is horrible, far worse than the behavior you would get on Android 5.1 and earlier using `AlarmManager`. But, it's the ultimate workaround, which is why it is demonstrated in the `AlarmManager/AntiDoze` sample application.

The AntiDoze sample is based off of the greenrobot's EventBus sample from the chapter on event bus alternatives. In that app, we used `AlarmManager` to get control every 15 seconds to either update a fragment (if the UI was in the foreground) or show a `Notification` (if not). AntiDoze gets rid of the every-event `Notification`, replacing it with appending an entry to a log file. And, it replaces `AlarmManager` with `ScheduledExecutorService` inside of a foreground `Service`, trying to run forever and get control every 15 seconds along the way.

This app has two product flavors defined in its `app/build.gradle` file, `normal` and `foreground`:

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'org.greenrobot:eventbus:3.0.0'
    implementation "com.android.support:support-v13:27.0.2"
}

android {
    compileSdkVersion 27
    buildToolsVersion '27.0.3'

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 27
    }

    flavorDimensions "default"

    productFlavors {
        foreground {
            dimension "default"
            buildConfigField "boolean", "IS_FOREGROUND", "true"
        }

        normal {
            dimension "default"
            buildConfigField "boolean", "IS_FOREGROUND", "false"
        }
    }
}
```

(from AlarmManager/AntiDoze/app/build.gradle)

A `normal` build will use a regular `Service`; a `foreground` build will use a foreground `Service`.

The launcher activity is `EventDemoActivity`. Its `onCreate()` method will do three things:

1. If we are on Android 6.0 or higher, it will use `isIgnoringBatteryOptimizations()` on `PowerManager` to see if we are already on the battery optimization whitelist, and if not, display a system-

supplied dialog-themed activity to ask the user to add our app to the whitelist
2. If we do not already have the `EventLogFragment`, add it
3. If we do not already have the `EventLogFragment`, also start up the `ScheduledService`, as probably it is not already running

```
package com.commonsware.android.antidoze;

import android.support.v4.app.FragmentActivity;
import android.content.Intent;
import android.net.Uri;
import android.os.Build;
import android.os.Bundle;
import android.os.PowerManager;
import android.provider.Settings;

public class EventDemoActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (Build.VERSION.SDK_INT>Build.VERSION_CODES.LOLLIPOP_MR1) {
      String pkg=getPackageName();
      PowerManager pm=getSystemService(PowerManager.class);

      if (!pm.isIgnoringBatteryOptimizations(pkg)) {
        Intent i=
          new Intent(Settings.ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS)
            .setData(Uri.parse("package:"+pkg));

        startActivity(i);
      }
    }

    if (getSupportFragmentManager().findFragmentById(android.R.id.content)==null) {
      getSupportFragmentManager().beginTransaction()
        .add(android.R.id.content,
          new EventLogFragment()).commit();
      startService(new Intent(this, ScheduledService.class));
    }
  }
}
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/EventDemoActivity.java)

To be able to use `ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`, we need to request and hold the `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS` permission, which we handle in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  package="com.commonsware.android.antidoze"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">
```

```
  <uses-permission android:name="android.permission.WAKE_LOCK"/>
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
  <uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name="EventDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <receiver android:name="PollReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>

    <receiver android:name="StopReceiver"/>

    <service android:name="ScheduledService"/>
  </application>

</manifest>
```

(from AlarmManager/AntiDoze/app/src/main/AndroidManifest.xml)

The rest of the UI layer is unchanged. Where the differences really creep in is with `ScheduledService`. This used to be a `WakefulIntentService`, triggered by an alarm event. Now, it is a regular service, designed to run all the time.

As part of initializing the `ScheduledService` class, we create an instance of `ScheduledExecutorService`, through the `newSingleThreadScheduledExecutor()` static method on the `Executors` utility class:

```
  private ScheduledExecutorService sched=
    Executors.newSingleThreadScheduledExecutor();
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/ScheduledService.java)

In `onCreate()`, we:

- Acquire a partial wakelock

- Call a private `foregroundify()` method to make our service be a foreground service with a suitable `Notification`, if our `IS_FOREGROUND` value is `true` based upon on our product flavor
- Set up a `File` for use with logging (named `log`), including creating the directory for it if needed
- Call `scheduleAtFixedRate()` on the `ScheduledExecutorService` to get control every 15 seconds

```
@Override
public void onCreate() {
  super.onCreate();

  PowerManager mgr=(PowerManager)getSystemService(POWER_SERVICE);

  wakeLock=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
    getClass().getSimpleName());
  wakeLock.acquire();

  if (BuildConfig.IS_FOREGROUND) {
    foregroundify();
  }

  log=new File(getExternalFilesDir(null), "antidoze-log.txt");
  log.getParentFile().mkdirs();
  sched.scheduleAtFixedRate(this, 0, 15, TimeUnit.SECONDS);
}
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/ScheduledService.java)

We can pass the service itself to `scheduleAtFixedRate()` because it implements the `Runnable` interface. Its `run()` method uses greenrobot's EventBus to tell the UI layer about our event, plus it calls an `append()` method to log that event to our `log` file:

```
@Override
public void run() {
  RandomEvent event=new RandomEvent(rng.nextInt());

  EventBus.getDefault().post(event);
  append(log, event);
}
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/ScheduledService.java)

`append()` simply uses Java file I/O to append a line to the `log` file:

```
private void append(File f, RandomEvent event) {
  try {
    FileOutputStream fos=new FileOutputStream(f, true);
    Writer osw=new OutputStreamWriter(fos);

    osw.write(event.when.toString());
    osw.write(" : ");
    osw.write(Integer.toHexString(event.value));
    osw.write('\n');
    osw.flush();
    fos.flush();
    fos.getFD().sync();
    fos.close();

    Log.d(getClass().getSimpleName(),
      "logged to "+f.getAbsolutePath());
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(),
      "Exception writing to file", e);
  }
}
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/ScheduledService.java)

The `foregroundify()` method, called from `onCreate()`, creates a `Notification` and calls `startForeground()` to make the service be a foreground service:

```
private void foregroundify() {
  NotificationManager mgr=
    (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
  }

  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);
  Intent iActivity=new Intent(this, EventDemoActivity.class);
  PendingIntent piActivity=
    PendingIntent.getActivity(this, 0, iActivity, 0);
  Intent iReceiver=new Intent(this, StopReceiver.class);
  PendingIntent piReceiver=
    PendingIntent.getBroadcast(this, 0, iReceiver, 0);

  b.setAutoCancel(true)
```

```
      .setDefaults(Notification.DEFAULT_ALL)
      .setContentTitle(getString(R.string.app_name))
      .setContentIntent(piActivity)
      .setSmallIcon(R.drawable.ic_launcher)
      .setTicker(getString(R.string.app_name))
      .addAction(R.drawable.ic_stop_white_24dp,
        getString(R.string.notif_stop),
        piReceiver);

    startForeground(NOTIFY_ID, b.build());
  }
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/ScheduledService.java)

The `Notification` includes a "stop" action, pointing to a `StopReceiver`, which just uses `stopService()` to stop the service. This allows the user to shut down our background service at any point, just via the `Notification`.

When the service is stopped, `onDestroy()` tidies things up, notably releasing the wakelock:

```
  @Override
  public void onDestroy() {
    sched.shutdownNow();
    wakeLock.release();
    stopForeground(true);

    super.onDestroy();
  }
```

(from AlarmManager/AntiDoze/app/src/main/java/com/commonsware/android/antidoze/ScheduledService.java)

Running this overnight on an Android 6.0 device shows that, indeed, we get control every 15 seconds, as desired. The device's battery drains commensurately, considering that we are keeping the CPU powered on all of the time. Either the whitelist keeps us going (`normal` flavor) or the foreground service keeps us going (`foreground` flavor).

setAlarmClock()

`AlarmManager` also has a `setAlarmClock()` method, added in API Level 21. This works a bit like `setExact()` (and, hence, `setExactAndAllowWhileIdle()`), in that you provide a time to get control and a `PendingIntent` to be invoked at that time. From the standpoint of power management, Doze mode leaves `setAlarmClock()` events alone, and so they are executed at the appropriate time regardless of device

state. However, at the same time, `setAlarmClock()` has some user-visible impacts that make it suitable for certain apps (e.g., calendar reminders) and unsuitable for others (e.g., polling).

Hope Somebody Else Does Something

Doze mode is for the entire device. Hence, your app may wind up getting control more frequently than you might expect, even without any code changes, simply because *somebody else* is doing something to get control more frequently.

Scheduling Content Monitoring

One long-standing challenge in Android is finding out when content changes in other apps. While `ContentObserver` is great for this purpose, you have to have a running process for it to work. As a result, some apps try desperately to keep a process running all the time to find out about changes to foreign `ContentProviders`, tying up system RAM as a result.

`JobScheduler`, as of Android 7.0, has an option to effectively register a `ContentObserver` for you. You indicate the `Uri` to monitor, and it invokes your `JobService` when the data at that `Uri` changes. This way, you do not need to keep a process around.

To do that, you create a `JobInfo.TriggerContentUri` object, identifying what to monitor. You pass that to `addTriggerContentUri()` on your `JobInfo.Builder`, and schedule the resulting `JobInfo` with the `JobScheduler` as before.

For example, the `JobScheduler/Content` sample project asks `JobScheduler` to monitor the `ContactsContract` provider for new contacts.

`MainActivity` has virtually nothing to do with any of this, but instead goes through all the work to set up runtime permission access to the `READ_CONTACTS` permission:

```
package com.commonsware.android.jobsched.content;

import android.app.Activity;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.support.v4.app.ActivityCompat;
import android.support.v4.content.ContextCompat;
import android.view.View;
```

```
import android.widget.Toast;
import static android.Manifest.permission.READ_CONTACTS;

public class MainActivity extends Activity {
  private static final String[] PERMS_ALL={
    READ_CONTACTS
  };
  private static final int RESULT_PERMS_INITIAL=1339;
  private static final String STATE_IN_PERMISSION=
    "com.commonsware.android.jobsched.content.inPermission";
  private boolean isInPermission=false;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (savedInstanceState!=null) {
      isInPermission=
        savedInstanceState.getBoolean(STATE_IN_PERMISSION, false);
    }

    if (!isInPermission) {
      if (hasPermission(READ_CONTACTS)) {
        configureJob();
      }
      else {
        isInPermission=true;
        ActivityCompat.requestPermissions(this, PERMS_ALL,
          RESULT_PERMS_INITIAL);
      }
    }
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putBoolean(STATE_IN_PERMISSION, isInPermission);
  }

  @Override
  public void onRequestPermissionsResult(int requestCode,
                                         String[] permissions,
                                         int[] grantResults) {
    boolean sadTrombone=true;

    isInPermission=false;
```

```
    if (requestCode==RESULT_PERMS_INITIAL) {
      if (hasPermission(READ_CONTACTS)) {
        configureJob();
        sadTrombone=false;
      }
    }

    if (sadTrombone) {
      Toast.makeText(this, R.string.msg_no_perm,
        Toast.LENGTH_LONG).show();
    }
  }

  private void configureJob() {
    Toast.makeText(this, R.string.msg_add,
      Toast.LENGTH_LONG).show();
    DemoJobService.schedule(this);
    finish();
  }

  private boolean hasPermission(String perm) {
    return(ContextCompat.checkSelfPermission(this, perm)==
      PackageManager.PERMISSION_GRANTED);
  }
}
```

(from JobScheduler/Content/app/src/main/java/com/commonsware/android/jobsched/content/MainActivity.java)

Eventually, though, if the user agrees to the permission, `MainActivity` calls a `static` `schedule()` method on `DemoJobService`, to set up the content monitor:

```
  static void schedule(Context ctxt) {
    ComponentName cn=
      new ComponentName(ctxt, DemoJobService.class);
    JobInfo.TriggerContentUri trigger=
      new JobInfo.TriggerContentUri(CONTENT_URI,
        JobInfo.TriggerContentUri.FLAG_NOTIFY_FOR_DESCENDANTS);
    JobInfo.Builder b=
      new JobInfo.Builder(ME_MYSELF_AND_I, cn)
        .addTriggerContentUri(trigger);
    JobScheduler jobScheduler=
      (JobScheduler)ctxt.getSystemService(Context.JOB_SCHEDULER_SERVICE);

    jobScheduler.schedule(b.build());
  }
```

(from JobScheduler/Content/app/src/main/java/com/commonsware/android/jobsched/content/DemoJobService.java)

Here, we:

- Create a `ComponentName` identifying our `JobService`
- Create a `TriggerContentUri`, asking for `ContactsContract.Contacts.CONTENT_URI` (imported via `import static`), and asking to be notified about changes in any "descendants" (i.e., already-existing contacts)
- Pass those two values, plus a job ID, to `JobInfo.Builder`
- Get a `JobScheduler` via `getSystemService()`
- Build the `JobInfo` and `schedule()` it with the `JobScheduler`

The rest of `DemoJobService` handles the results, in this case just raising a `Notification`:

```
@Override
public boolean onStartJob(JobParameters params) {
  NotificationManager mgr=
    (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
    mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
    mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
      "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
  }

  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER)
      .setAutoCancel(true)
      .setDefaults(Notification.DEFAULT_ALL)
      .setContentTitle("You added a contact!")
      .setSmallIcon(android.R.drawable.stat_notify_more);

  mgr.notify(NOTIFY_ID, b.build());

  return(false);
}

@Override
synchronized public boolean onStopJob(JobParameters params) {
  return(false);
}
```

(from JobScheduler/Content/app/src/main/java/com/commonsware/android/jobsched/content/DemoJobService.java)

However, if we wanted, the `JobParameters` passed into `onStartJob()` contains information about what changed.

`getTriggeredContentAuthorities()` returns a `String` array of the names of the authorities whose changes triggered this job, if any. It will return `null` if the job triggered for some other reason, such as a deadline.

If `getTriggeredContentAuthorities()` returns a non-`null` value, then you can try calling `getTriggeredContentUris()` to find out the specific `Uri` values that changed. However, this may be `null`, if there were too many changes to report (the limit is ~50).

Note that there are limitations on these content-monitoring jobs:

- They cannot be persisted, and so you need to re-request them after a reboot
- They cannot be periodic, though other job restrictions may still work (e.g., must be on a charger, must have a network connection)
- The job is a one-shot event — if you want continuous updates, you need to schedule a fresh job after this one is invoked (either with matches or due to hitting the deadline)

One problem with monitoring content for changes is that those changes may occur too frequently. In Android 7.0, you have two new `JobInfo.Builder` methods that you can use to manage this:

- `setTriggerContentUpdateDelay()` indicates how long after the last content change before the job will be invoked. For example, suppose that through some sort of sync operation, a provider that you are monitoring is updated 10 times within a second, then is quiet. By default, your job would be invoked 10 times. But, if you pass something like `3000` to `setTriggerContentUpdateDelay()`, your job would be invoked once, 3000 milliseconds after the last of that burst of updates.
- `setTriggerContentMaxDelay()` puts an upper bound for how long you are willing to wait before the job is invoked. If the provider is very busy, and your `setTriggerContentUpdateDelay()` counter keeps getting reset due to updates, it may be quite some time after the burst began before you finally have your job run. `setTriggerContentMaxDelay()` sets a limit for how long we will wait; if this time elapses, your job will be run even if updates are ongoing.

JobScheduler as Work Queue

Android 8.0 adds a work queue mode to `JobScheduler`. `IntentService` had such a queue, after a fashion, in that it would process one `Intent` at a time through `onHandleIntent()`, queuing up other `Intent` objects that arrive while `onHandleIntent()` is busy. `JobScheduler` now offers a similar capability for your `JobService`, where you can post jobs and have your `JobService` end when the work is completed.

The `JobScheduler/WorkQueue` sample project illustrates this. We have a `JobService` that will download files as its "work", posting the results on an event bus. Our client is an instrumentation test that will confirm that the downloads work as expected.

Defining Some "Work"

The "work" to be enqueued comes in the form of a `JobWorkItem`, which is a thin wrapper around an `Intent`. That `Intent` is used for payload, such as filling in extras. That `Intent` is *not* used for actually starting or binding to a service.

The sample project has a `WorkService` which is the `JobService` for handling these jobs. It has a static `buildWorkItem()` method that will help create a `JobWorkItem` from the two pieces of data that we want as our "work":

- A URL identifying something to download
- An `int` identifying which piece of work this is

```
public static JobWorkItem buildWorkItem(int workIndex, String url) {
  Intent i=new Intent();

  i.setData(Uri.parse(url));
  i.putExtra(EXTRA_WORK_INDEX, workIndex);

  return(new JobWorkItem(i));
}
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

Enqueuing the Work

`JobScheduler` in Android 8.0 has an `enqueue()` method. It takes a `JobInfo`, the way you normally schedule jobs, along with a `JobWorkItem`. It arranges to start the `JobService` if it is not already running and adds the `JobWorkItem` to its work queue.

One strong recommendation outlined in the documentation is to try to use the same (or an equivalent) `JobInfo` object for each `enqueue()` call. Changing the job characteristics — in particular, tightening constraints, like now needing to be on a charger — will cause Android to have to stop your running `JobService` (if it is running) and restart it, perhaps later.

`WorkService` has a static `enqueueWork()` method that handles all of the details:

```
public static JobInfo enqueueWork(Context ctxt, JobInfo jobInfo, List<JobWorkItem> work) {
  JobScheduler jobScheduler=ctxt.getSystemService(JobScheduler.class);

  if (jobInfo==null) {
    ComponentName cn=new ComponentName(ctxt, WorkService.class);

    jobInfo=new JobInfo.Builder(JOB_ID, cn)
      .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
      .build();
  }

  for (JobWorkItem item : work) {
    jobScheduler.enqueue(jobInfo, item);
  }

  return(jobInfo);
}
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

`enqueueWork()` takes three parameters:

- A `Context`
- A `JobInfo` returned from a previous `enqueueWork()` call, or `null` if we do not have one
- The instances of `JobWorkItem` to enqueue, in this case in the form of a `List`

`enqueueWork()` will create a `JobInfo` object if needed, but otherwise it will reuse the passed-in `JobInfo`. That `JobInfo` requires a network connection, as we will be downloading a file, but otherwise sets no constraints. Then, `enqueueWork()` simply iterates over the `JobWorkItem` objects and calls `enqueue()` to register each of them. `enqueueWork()` returns the `JobInfo` object that we created or used, for later reuse.

Working Off the Queue

As with any `JobService`, `onStartJob()` is our entry point for doing the work requested by whoever scheduled the job. In this case, it delegates the real work to a `scheduleWork()` method, then returns `true` to indicate that the work is ongoing.

```
@Override
public boolean onStartJob(JobParameters params) {
  scheduleWork(params);

  return(true);
}
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

`scheduleWork()`, in turn, calls `dequeueWork()` on the `JobParameters`, until it returns `null` to indicate that there is no more queued work:

```
private void scheduleWork(final JobParameters params) {
  if (!threadPool.isShutdown()) {
    JobWorkItem item;

    while ((item=params.dequeueWork())!=null) {
      final int workIndex=item.getIntent().getIntExtra(EXTRA_WORK_INDEX, -1);
      final String url=item.getIntent().getData().toString();
      final JobWorkItem itemToDo=item;

      threadPool.execute(new Runnable() {
        @Override
        public void run() {
          download(workIndex, url);
          params.completeWork(itemToDo);
          scheduleWork(params);
        }
      });
    }
  }
}
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

`dequeueWork()` returns a `JobWorkItem` object if there is one. We can retrieve values out of it and arrange for a `ThreadPoolExecutor` to perform the actual work. Remember: `onStartJob()` is called on the main application thread, so you cannot process the work directly in most cases.

That `ThreadPoolExecutor` is a simple field on the `WorkService`, initialized via `Executors.newFixedThreadPool()`:

```
private ExecutorService threadPool=Executors.newFixedThreadPool(3);
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

For each `JobWorkItem`, the `ThreadPoolExecutor` does three things on a background thread:

1. It calls a `download()` method to download the file identified by the supplied URL
2. It calls `completeWork()` on the `JobParameters`, to indicate that this work item is complete and can be removed from the queue
3. It calls `scheduleWork()` again, in case more jobs arrived while we were busy downloading the file

The result is that we keep calling `scheduleWork()` until we are out of `JobWorkItem` instances to process. At that point, the final `dequeueWork()` call will not only return `null` but also arrange to shut down our `JobService`. In particular, we do *not* call `jobFinished()` to do that ourselves.

`download()` uses OkHttp to download the content identified by the URL, then uses Okio's `HashingSource` to compute the SHA-256 hash of the content:

```
private void download(int workIndex, String url) {
  try {
    Response response=ok.newCall(new Request.Builder().url(url).build()).execute();
    HashingSource hashingSource=HashingSource.sha256(response.body().source());

    EventBus.getDefault().post(new Result(hashingSource.hash(), workIndex, null));
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(), "Exception from OkHttp", e);
    EventBus.getDefault().post(new Result(null, workIndex, e));
  }
}
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

`download()` posts the results — either the hash or an `Exception`, along with the `int` identifying this piece of work — on an event bus, wrapped in a `Result` object:

```
public static class Result {
  public final ByteString hash;
  public final int workIndex;
  public final Exception e;

  Result(ByteString hash, int workIndex, Exception e) {
    this.hash=hash;
    this.workIndex=workIndex;
    this.e=e;
  }
}
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

However, it is possible that while our downloads are going on, that we lose connectivity. Not only will OkHttp start throwing errors, but `JobScheduler` will trigger a call to `onStopJob()` on our `WorkService`. There, we just `shutdown()` the thread pool, as we do when all of the work is done:

```
  @Override
  public boolean onStopJob(JobParameters params) {
    threadPool.shutdown();

    return(true);
  }
```

(from JobScheduler/WorkQueue/app/src/main/java/com/commonsware/android/job/work/WorkService.java)

Testing the Service

The sample project has no significant UI — the `MainActivity` just shows a `Toast` telling you to run the instrumentation tests. There, you will find a `WorkTests` class that:

- Schedules two batches of work items, one second apart, to download the URL contents a random number of times
- Waits for all of the events to be received on the event bus
- Fails if either we get an exception or we missed a work item (based on its index)

```
@RunWith(AndroidJUnit4.class)
public class WorkTests {
  private static final String URL=
    "https://commonsware.com/Android/Android-1_0-CC.pdf";
  private static final String EXPECTED_HASH_HEX=
    "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855";
  private CountDownLatch latch;
  private Exception e=null;
  private HashSet<Integer> workIndices=new HashSet<>();

  @Test
  public void testWork() throws Exception {
    Random r=new Random();
    int firstBatchCount=4+r.nextInt(4);
    int secondBatchCount=4+r.nextInt(4);

    latch=new CountDownLatch(firstBatchCount+secondBatchCount);

    EventSink sink=new EventSink();

    EventBus.getDefault().register(sink);
```

```
    try {
      JobInfo jobInfo=null;
      ArrayList<JobWorkItem> items=new ArrayList<>();

      for (int i=0;i<firstBatchCount;i++) {
        items.add(WorkService.buildWorkItem(i, URL));
      }

      jobInfo=WorkService.enqueueWork(InstrumentationRegistry.getTargetContext(),
        jobInfo, items);

      SystemClock.sleep(1000);

      items.clear();

      for (int i=0;i<secondBatchCount;i++) {
        items.add(WorkService.buildWorkItem(i+firstBatchCount, URL));
      }

      WorkService.enqueueWork(InstrumentationRegistry.getTargetContext(),
        jobInfo, items);

      latch.await(firstBatchCount+secondBatchCount, TimeUnit.SECONDS);

      if (e!=null) {
        throw e;
      }
    }
    finally {
      EventBus.getDefault().unregister(sink);
    }

    assertEquals(firstBatchCount+secondBatchCount, workIndices.size());
  }

  private class EventSink {
    @Subscribe(threadMode =ThreadMode.ASYNC)
    public void onWorkResult(WorkService.Result result) {
      workIndices.add(result.workIndex);

      if (result.e!=null) {
        WorkTests.this.e=result.e;
      }
      else  {
        String hash=result.hash.hex();

        if (!EXPECTED_HASH_HEX.equals(hash)) {
          WorkTests.this.e=
            new IllegalStateException(String.format("Expected hash of %s, received %s",
              EXPECTED_HASH_HEX, hash));
        }
      }

      latch.countDown();
    }
  }
}
```

(from JobScheduler/WorkQueue/app/src/androidTest/java/com/commonsware/android/job/work/test/WorkTests.java)

Work Limits

As noted previously, the work queue system with `JobScheduler` relies on your using the same or an equivalent `JobInfo` for each piece of work. Otherwise, our `JobService` needs to stop processing jobs and restart them, perhaps after some delay.

Trail: Hardware and System Services

Accessing Location-Based Services

A popular feature on current-era mobile devices is GPS capability, so the device can tell you where you are at any point in time. While the most popular use of GPS service is mapping and directions, there are other things you can do if you know your location. For example, you might set up a dynamic chat application where the people you can chat with are based on physical location, so you are chatting with those you are nearest. Or, you could automatically "geotag" posts to Twitter or similar services.

GPS is not the only way a mobile device can identify your location. Alternatives include:

1. Cell tower triangulation, where your position is determined based on signal strength to nearby cell towers
2. Proximity to public WiFi "hotspots" that have known geographic locations
3. GPS alternatives, such as GLONASS (Russia), Galileo (European Union, still under development), and Compass (China, still under development)

Android devices may have one or more of these services available to them. You, as a developer, can ask the device for your location, plus details on what providers are available. There are even ways for you to simulate your location in the emulator, for use in testing your location-enabled applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on threads.

Location Providers: They Know Where You're Hiding

Android devices can have access to several different means of determining your location. Some will have better accuracy than others. Some may be free, while others may have a cost associated with them. Some may be able to tell you more than just your current position, such as your elevation over sea level, or your current speed.

Android, therefore, has abstracted all this out into a set of `LocationProvider` objects. Your Android environment will have zero or more `LocationProvider` instances, one for each distinct locating service that is available on the device. Providers know not only your location, but also their own characteristics, in terms of accuracy, cost, etc. There are two main providers: `GPS_PROVIDER` (which uses GPS) and `NETWORK_PROVIDER` (which uses cell tower triangulation and WiFi hotspot proximity).

You, as a developer, will use a `LocationManager`, which holds the `LocationProvider` set, to figure out which `LocationProvider` is right for your particular circumstance. You will also need a permission in your application, or the various location APIs will fail due to a security violation.

Depending on which location providers you wish to use, you may need `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. Note that `ACCESS_COARSE_LOCATION` may intentionally "fuzz" or filter out location fixes that are "too good" (i.e., more accurate than a city block), such as those obtained from being near a known WiFi hotspot. The `GPS_PROVIDER` specifically requires `ACCESS_FINE_LOCATION` to work, at least on modern versions of Android. Also note that these permissions are `dangerous`, and therefore if your `targetSdkVersion` is 23 or higher, you need to ask for these permissions at runtime.

Finding Yourself

The obvious thing to do with a location service is to figure out where you are right now.

To do that, you need to get a `LocationManager` — call `getSystemService(LOCATION_SERVICE)` from your activity or service and cast it to be a `LocationManager`.

The next step to find out where you are is to get the name of the `LocationProvider` you want to use. Here, you have two main options:

- Ask the user to pick a provider
- Find the best-match provider based on a set of criteria

If you want the user to pick a provider, calling `getProviders()` on the `LocationManager` will give you a `List` of providers, which you can then present to the user for selection.

Or, you can create and populate a `Criteria` object, stating the particulars of what you want out of a `LocationProvider`, such as:

1. `setAltitudeRequired()` to indicate if you need the current altitude or not
2. `setAccuracy()` to set a minimum level of accuracy, in meters, for the position
3. `setCostAllowed()` to control if the provider must be free or if it can incur a cost on behalf of the device user

Given a filled-in `Criteria` object, call `getBestProvider()` on your `LocationManager`, and Android will sift through the criteria and give you the best answer. Note that not all of your criteria may be met – all but the monetary cost criterion might be relaxed if nothing matches.

You are also welcome to hard-wire in a `LocationProvider` name (e.g., `GPS_PROVIDER`), perhaps just for testing purposes.

Once you know the name of the `LocationProvider`, you can call `getLastKnownLocation()` to find out where you were recently. However, unless something else is causing the desired provider to collect fixes (e.g., unless the GPS radio is on), `getLastKnownLocation()` will return `null`, indicating that there is no known position. On the other hand, `getLastKnownLocation()` incurs no monetary or power cost, since the provider does not need to be activated to get the value.

This method returns a `Location` object, which can give you the latitude and longitude of the device in degrees as a Java `double`. If the particular location provider offers other data, you can get at that as well:

1. For altitude, `hasAltitude()` will tell you if there is an altitude value, and `getAltitude()` will return the altitude in meters.

2. For bearing (i.e., compass-style direction), `hasBearing()` will tell you if there is a bearing available, and `getBearing()` will return it as degrees east of true north.
3. For speed, `hasSpeed()` will tell you if the speed is known and `getSpeed()` will return the speed in meters per second.

A more likely approach to getting the `Location` from a `LocationProvider`, though, is to register for updates, as described in the next section.

On the Move

Not all location providers are necessarily immediately responsive. GPS, for example, requires activating a radio and getting a fix from the satellites before you get a location. That is why Android does not offer a `getMeMyCurrentLocationNow()` method. Combine that with the fact that your users may well want their movements to be reflected in your application, and you are probably best off registering for location updates and using that as your means of getting the current location.

The `Location/Classic` sample application shows how to register for updates and use them when they arrive. It also shows how to deal with the runtime permissions that we need for locations.

Getting Permission

Our UI is implemented in `MainActivity` and its associated `WeatherFragment`. However, `MainActivity` extends `AbstractPermissionActivity`, which handles the basics of ensuring that we have the `ACCESS_FINE_LOCATION` permission that our app needs in order to get a location fix. This is a variation on the `AbstractPermissionActivity` covered in the material on runtime permissions earlier in the book.

Subclasses of `AbstractPermissionActivity` need to implement three methods:

- `getDesiredPermissions()`, returning the array of permission names that the activity needs in order to proceed
- `onReady()`, called by `AbstractPermissionActivity` once we get all of the requested permissions
- `onPermissionDenied()`, called by `AbstractPermissionActivity` if the user did not grant us all of the requested permissions when we asked for them

In the case of `MainActivity`, `getDesiredPermissions()` asks for `ACCESS_FINE_LOCATION`, `onReady()` displays the `WeatherFragment`, and `onPermissionDenied()` shows a `Toast` and finishes the activity:

```
package com.commonsware.android.weather2;

import android.Manifest;
import android.widget.Toast;

public class MainActivity extends AbstractPermissionActivity {
  private static final String[] PERMS=
    {Manifest.permission.ACCESS_FINE_LOCATION};

  @Override
  protected String[] getDesiredPermissions() {
    return(PERMS);
  }

  @Override
  protected void onPermissionDenied() {
    Toast
      .makeText(this, R.string.msg_no_perm, Toast.LENGTH_LONG)
      .show();
    finish();
  }

  @Override
  protected void onReady() {
    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
        .add(android.R.id.content,
          new WeatherFragment()).commit();
    }
  }
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/MainActivity.java)

The result is that by the time `WeatherFragment` is displayed, we now have `ACCESS_FINE_LOCATION`, and it is safe for us to use `LocationManager`.

Modelling the Weather

This app will use the US National Weather Service (NWS) to get weather details for a given location. This may not work outside the US. And even inside the US, the

National Weather Service's servers seem to have issues sometimes. But, it is free, with no registration or API keys required, at least at the present time.

The NWS REST Web service serves data in a variety of formats, including JSON. The `WeatherResponse` POJO models the subset of that JSON that we need for displaying basic weather details to the user:

```
package com.commonsware.android.weather2;

import java.util.List;

public class WeatherResponse {
  public final Properties properties=null;

  public static class Properties {
    public final List<Period> periods=null;
  }

  public static class Period {
    public final String startTime=null;
    public final int temperature;
    public final String temperatureUnit=null;
    public final String icon=null;

    public Period() {
      temperature=0;
    }
  }
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/WeatherResponse.java)

The JSON object that gets returned has a `properties` field, which in turn has a `periods` field. The `periods` are a `List` of `Period` objects, each of which has the time for the forecast, the projected temperature (and unit of measure, such as "F" for Fahrenheit), and a URL to an icon representing the type of weather (sunny, cloudy, rain, snow, zombie invasion, etc.).

Requesting Updates

In `onCreate()` of `WeatherFragment`, we get our hands on a `LocationManager` via a call to `getSystemService()`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setRetainInstance(true);

    mgr=(LocationManager)getActivity()
      .getSystemService(Context.LOCATION_SERVICE);

    Retrofit retrofit=
      new Retrofit.Builder()
        .baseUrl("https://api.weather.gov")
        .addConverterFactory(GsonConverterFactory.create())
        .build();

    nws=retrofit.create(NWSInterface.class);
  }
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/WeatherFragment.java)

We also create a `Retrofit` instance, pointing to the API endpoint of the NWS. Retrofit is a library for accessing REST-style Web services, and it is profiled earlier in the book.

We then use the `Retrofit` instance to create an instance of `NWSInterface`. This interface is augmented with Retrofit annotations to describe how we should request a weather forecast:

```
package com.commonsware.android.weather2;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Headers;
import retrofit2.http.Path;

public interface NWSInterface {
  @Headers("Accept: application/geo+json")
  @GET("/points/{lat},{lon}/forecast")
  Call<WeatherResponse> getForecast(@Path("lat") double latitude,
                                    @Path("lon") double longitude);
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/NWSInterface.java)

Here, we have a `getForecast()` method that takes a latitude and longitude and pours them into the path portion of the URL for our REST request. We also indicate that we want a GeoJSON response (`application/geo+json`); `WeatherResponse` models a small bit of what a GeoJSON response might contain.

In `onStart()`, we request location updates:

```
@Override
@SuppressWarnings({"MissingPermission"})
public void onStart() {
  super.onStart();

  mgr.requestLocationUpdates(LocationManager.GPS_PROVIDER, 3600000,
    1000, this);
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/WeatherFragment.java)

`requestLocationUpdates()` on `LocationManager` takes four parameters:

- The name of the location provider you wish to use
- How long, in milliseconds, *should* have elapsed before we might get a location update
- How far, in meters, must the device have moved before we might get a location update
- An implementation of the `LocationListener` interface that will be notified of key location-related events

In our case, we are asking for updates from the `GPS_PROVIDER` once an hour, where the device has moved a least 1 kilometer, with our fragment serving as the `LocationListener` implementation.

Bear in mind that the time parameter is only a guide to help steer Android from a power consumption standpoint. You may get many more location updates than this. To get the maximum number of location updates, supply `0` for both the time and distance constraints.

In `onStop()`, we call `removeUpdates()`, so we only get location updates while we are visible:

```
@Override
@SuppressWarnings({"MissingPermission"})
public void onStop() {
  mgr.removeUpdates(this);

  super.onStop();
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/WeatherFragment.java)

The `@SuppressWarnings({"MissingPermission"})` annotation on `onStart()` and `onStop()` are because Android Studio cannot determine for certain that we have implemented runtime permissions properly. Since the IDE does not know if we hold `ACCESS_FINE_LOCATION`, it complains. However, we *have* implemented runtime permissions — this fragment will not exist until we have that permission. So, we suppress the Lint warning.

Implementing the Listener

`LocationListener` requires four methods, the big one being `onLocationChanged()`, where you will receive your `Location` object when an update is ready:

```
@Override
public void onLocationChanged(Location location) {
  double roundedLat=(double)Math.round(location.getLatitude()*10000d)/10000d;
  double roundedLon=(double)Math.round(location.getLongitude()*10000d)/10000d;

  nws.getForecast(roundedLat, roundedLon)
    .enqueue(new Callback<WeatherResponse>() {
      @Override
      public void onResponse(Call<WeatherResponse> call,
                             Response<WeatherResponse> response) {
        if (response.code()==200) {
          adapter=new ForecastAdapter(response.body().properties.periods);
          setListAdapter(adapter);
        }
        else {
          Toast.makeText(getActivity(), R.string.msg_nws,
            Toast.LENGTH_LONG).show();
        }
      }

      @Override
      public void onFailure(Call<WeatherResponse> call, Throwable t) {
        Toast.makeText(getActivity(), t.getMessage(),
          Toast.LENGTH_LONG).show();
        Log.e(getClass().getSimpleName(),
          "Exception from Retrofit request to National Weather Service", t);
      }
    });
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/WeatherFragment.java)

Here, we:

- Round the location to four decimal places, as the NWS REST Web service will not accept more precise values than this and fails with an HTTP 500 response if you provide more decimal places

- Call the `getForecast()` on our Retrofit-supplied `NWSInterface` and `enqueue()` the resulting `Call`, so the network I/O is done on a background thread
- In `onResponse()`, if we got an HTTP 200 response from the Web service, we pass that `List` of `Period` objects to a `ForecastAdapter` and attach it to the `ListView`
- In `onResponse()` shows that we got some error from the Web service, or if there was a problem that triggered `onFailure()` (e.g., no Internet connection), we show a `Toast`

Displaying the Results

`ForecastAdapter` shows the time and temperature directly, by updating the associated `TextView` widgets. It delegates the icon-loading process to Picasso, discussed back in the chapter on Internet access.

```
private class ForecastAdapter extends ArrayAdapter<WeatherResponse.Period> {
  private int size;
  private java.text.DateFormat dateFormat;
  private java.text.DateFormat timeFormat;

  ForecastAdapter(List<WeatherResponse.Period> items) {
    super(getActivity(), R.layout.row, R.id.date, items);

    size=getActivity()
      .getResources()
      .getDimensionPixelSize(R.dimen.icon);
    dateFormat=DateFormat.getDateFormat(getActivity());
    timeFormat=DateFormat.getTimeFormat(getActivity());
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    WeatherResponse.Period item=getItem(position);

    if (!TextUtils.isEmpty(item.icon)) {
      ImageView icon=row.findViewById(R.id.icon);

      Picasso.with(getActivity()).load(item.icon)
        .resize(size, size).centerCrop().into(icon);
    }

    TextView title=row.findViewById(R.id.date);
```

```
    try {
      Date parsedStartTime=ISO8601.parse(item.startTime);
      String date=dateFormat.format(parsedStartTime);
      String time=timeFormat.format(parsedStartTime);

      title.setText(date+" "+time);
    }
    catch (ParseException e) {
      title.setText(item.startTime);
    }

    TextView temp=row.findViewById(R.id.temp);

    temp.setText(getString(R.string.temp, item.temperature,
      item.temperatureUnit));

    return(row);
  }
}
```

(from Location/Classic/app/src/main/java/com/commonsware/android/weather2/WeatherFragment.java)

The result is a `ListView` showing the weather forecast... at least if your location is somewhere covered by the US National Weather Service:

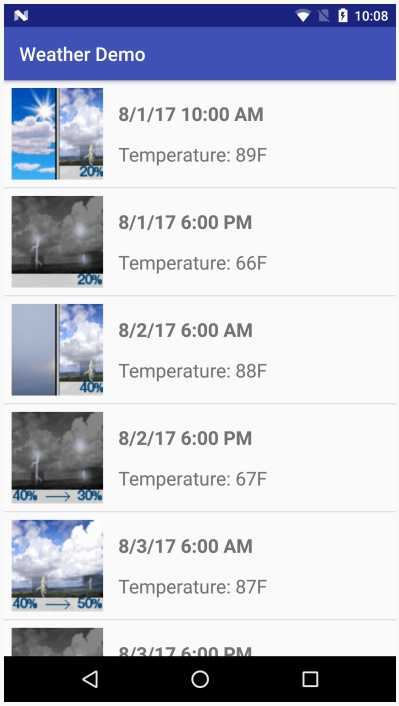


Figure 865: Weather in Eastern Pennsylvania

Getting Locations via PendingIntent

There is another version of `requestLocationUpdates()` that takes a `PendingIntent` rather than a `LocationListener`. This is useful if you want to be notified of changes in your position even when your code is not running. For example, if you are logging movements, you could use a `PendingIntent` that triggers a `BroadcastReceiver` (`getBroadcast()`) and have the `BroadcastReceiver` add the entry to the log. This way, your code is only in memory when the position changes, so you do not tie up system resources while the device is not moving.

Are We There Yet? Are We There Yet? Are We There Yet?

Sometimes, you want to know not where you are now, or even when you move, but when you get to where you are going. This could be an end destination, or it could be getting to the next step on a set of directions, so you can give the user the next turn.

To accomplish this, `LocationManager` offers `addProximityAlert()`. This registers a `PendingIntent`, which will be fired off when the device gets within a certain distance of a certain location. The `addProximityAlert()` method takes, as parameters:

1. The latitude and longitude of the position that you are interested in
2. A radius, specifying how close you should be to that position for the `Intent` to be raised
3. A duration for the registration, in milliseconds — after this period, the registration automatically lapses. A value of `-1` means the registration lasts until you manually remove it via `removeProximityAlert()`.
4. The `PendingIntent` to be raised when the device is within the "target zone" expressed by the position and radius

Note that it is not guaranteed that you will actually receive an `Intent`, if there is an interruption in location services, or if the device is not in the target zone during the period of time the proximity alert is active. For example, if the position is off by a bit, and the radius is a little too tight, the device might only skirt the edge of the target zone, or go by so quickly that the device's location isn't sampled while in the target zone.

It is up to you to arrange for an activity or receiver to respond to the `Intent` you register with the proximity alert. What you then do when the `Intent` arrives is up to you: set up a notification (e.g., vibrate the device), log the information to a content provider, post a message to a Web site, etc. Note that you will receive the `Intent` whenever the position is sampled and you are within the target zone – not just upon entering the zone. Hence, you will get the `Intent` several times, perhaps quite a few times depending on the size of the target zone and the speed of the device's movement.

Testing… Testing…

The Android emulator does not have the ability to get a fix from GPS, triangulate your position from cell towers, or identify your location by some nearby WiFi signal. So, if you want to simulate a moving device, you will need to have some means of providing mock location data to the emulator.

You can send location fixes via `telnet` to an emulator. The port number is in your emulator's title bar (usually `5554` for the first running emulator instance). You can then run:

```
telnet localhost 5554
```

to access the Android Console within the emulator. Running the `geo fix NNN NNN` command, where `NNN NNN` is your latitude and longitude, will have the emulator respond as if those coordinates came from GPS.

Alternative Flavors of Updates

There are more ways to get updates from `LocationManager` than the versions of `requestLocationUpdates()` we have seen so far. There are four major axes of difference:

1. Some versions of `requestLocationUpdates()` take a `Criteria` object, having Android give you fixes based on the best-available provider given the requirements stipulated in the `Criteria`
2. Some versions of `requestLocationUpdates()` take a `Looper` as a parameter, allowing you to receive updates on a background `HandlerThread` instead of the main application thread
3. Some versions of `requestLocationUpdates()` take a `PendingIntent` which will be executed, instead of calling your `LocationListener`
4. There are a few flavors of `requestSingleUpdate()`, which, as the name suggests, gives you just one location fix, rather than a stream until you remove the request for updates

For the `Criteria`-flavored versions of `requestLocationUpdates()` and `requestSingleUpdate()`, bear in mind that your code will still crash if there are no possible providers for your `Criteria`. For example, even if you use an empty `Criteria` object (for maximum possible matches), but GPS is disabled and the device lacks telephony (e.g., a tablet), you can get a crash like this one:

```
02-09 13:29:21.549: E/AndroidRuntime(2236): FATAL EXCEPTION: main
02-09 13:29:21.549: E/AndroidRuntime(2236): java.lang.RuntimeException: Unable to
resume activity {com.commonsware.android.mapsv2.location/
com.commonsware.android.mapsv2.location.MainActivity}:
java.lang.IllegalArgumentException: no providers found for criteria
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.performResumeActivity(ActivityThread.java:2564)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.handleResumeActivity(ActivityThread.java:2607)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2088)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
```

```
android.app.ActivityThread.access$600(ActivityThread.java:134)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1233)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Handler.dispatchMessage(Handler.java:99)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Looper.loop(Looper.java:137)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.main(ActivityThread.java:4699)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
java.lang.reflect.Method.invokeNative(Native Method)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
java.lang.reflect.Method.invoke(Method.java:511)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:787)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:554)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
dalvik.system.NativeStart.main(Native Method)
02-09 13:29:21.549: E/AndroidRuntime(2236): Caused by:
java.lang.IllegalArgumentException: no providers found for criteria
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Parcel.readException(Parcel.java:1331)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Parcel.readException(Parcel.java:1281)
02-09 13:29:21.549: E/AndroidRuntime(2236):   ... 19 more
```

Hence, you will still want to use `getProviders()` or `getBestProvider()` to ensure that your `Criteria` will resolve to *something* before you try using the `Criteria` to actually request fixes.

The Fused Option

Google Play Services — the proprietary API set supported by many Android devices – offers a fused location provider that simplifies location tracking. This capability is covered in the next chapter.

Locations and Features

Sometimes, requesting a permission implies that your app requires a certain hardware feature. For example, having a `<uses-permission>` element for the `CAMERA` permission implies that your app requires a camera. To undo that requirement, you

use a <uses-feature> element, with android:required="false" to stipulate that you do not necessarily *need* the hardware, though you may use it if it is available:

```
<uses-feature android:name="android.hardware.camera" android:required="false" />
```

When it comes to the location permissions, the behavior depends on your targetSdkVersion:

Permission	targetSdkVersion	Implied Feature Requirement
ACCESS_FINE_LOCATION	<=20	android.hardware.location and android.hardware.location.gps
ACCESS_FINE_LOCATION	>=21	android.hardware.location
ACCESS_COARSE_LOCATION	<=20	android.hardware.location and android.hardware.location.network
ACCESS_COARSE_LOCATION	>=21	android.hardware.location

In other words, a device with any sort of location technology will be able to install your app if you request ACCESS_FINE_LOCATION, not necessarily one with GPS capability.

This leads to two directions for applying <uses-feature> elements to control this behavior:

- If you can live without location technology, consider having a <uses-feature> element to say that android.hardware.location is not required
- If you want to ensure that the device has GPS capability, add a <uses-feature> element to say that android.hardware.location.gps *is* required

The Fused Location Provider

At the 2013 Google I|O conference, Google announced an update to Google Play Services that offers a "fused location provider", one that seamlessly uses all available location data to give you as accurate of a location as possible, as quickly as possible, with as little power consumption as possible. This serves as an adjunct to the traditional `LocationManager` approach for finding one's position. The fused location provider has a different API, though one that is similar in some respects to the `LocationManager` API.

In this chapter, we will examine how to use the fused location provider.

Prerequisites

This chapter assumes that you have read the preceding chapter on location-based services, along with that chapter's prerequisites.

Why Use the Fused Location Provider?

The traditional recipes for using location providers are a bit complicated, if you want to maximize results. Simply asking for a GPS fix is not that hard, but:

- What if GPS is disabled?
- What if GPS signals are unavailable (e.g., the device is indoors)?
- What about the GPS power drain?

The fused location provider is designed to address these sorts of concerns. Its implementation will blend data from GPS, cell tower triangulation, and WiFi hotspot proximity to determine the device's location, without your having to

manually set all of that up. The fused location provider will also take advantage of sensor data, so it does not try to update your location as frequently if the accelerometer indicates that you are not moving.

The net result is better location data, delivered more quickly, with (reportedly) less power consumption.

Why Not Use the Fused Location Provider?

The fused location provider is part of Google Play Services. Google Play Services is available on hundreds of millions of Android devices. However:

- It is closed source, and so we do not know what the Play Services all do, and whether anything that it does might be detrimental.
- It is proprietary, and so Play Services will not be available on the Kindle Fire series and other devices working solely from the Android open source project.
- Play Services is only available on devices that have the Play Store, as opposed to the old Android Market, and so older devices (e.g., Android 2.2 and older) are far less likely to have Play Services available.

If you are aiming to distribute your app solely through the Play Store, relying upon the Play Services framework is reasonable. If, however, you are distributing through other channels, you will either need to *conditionally* use the fused location provider on devices that offer it, or *avoid* the fused location provider entirely, falling back to the traditional `LocationManager` solution.

Finding Our Location, Once

The fused location provider requires a fair bit of setup, because of its dependence upon the Play Services framework. However, once that is established, the fused location provider is as easy to use, if not easier, than is `LocationManager`.

This section will review the `Location/FusedNew` sample application, which is a clone of the `Location/Classic` sample application from the previous chapter, revised to use the fused location provider to get a one-off weather forecast.

Installing and Attaching Google Play Services

If you have not done so already (e.g., for Maps V2), you will need to install the Play Services framework in your development environment.

Android Studio users should install the "Google Repository" entry in the SDK Manager. At that point, you can add a dependency upon the `com.google.android.gms:play-services-location` artifact for some appropriate version, such as `com.google.android.gms:play-services-location:7.8.0`.

Checking for Google Play Services

There is a fair bit of programming overhead to check for whether or not the Play Services Framework exists on the user's device and is up to date. Much of this will be the same for any app that uses Play Services, particularly for apps that use the `GoogleApiClient` as we will here.

The chapter on Play Services has an extensive section covering this overhead.

Permissions

To use the fused location provider, you still need the `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permissions. If you only hold `ACCESS_COARSE_LOCATION`, the data you get back will be limited to data that is sufficiently "fuzzy". Typically, if you are bothering using this provider, you will request `ACCESS_FINE_LOCATION` — if coarse location data is all you need, using `LocationManager` should be just as good and is compatible with more devices.

For Android 6.0+ devices, if your `targetSdkVersion` is 23 or higher, you are going to need to deal with the runtime permissions model. Both `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` are considered to be `dangerous` permissions. You will need both the `<uses-permission>` elements *and* specifically ask for user permission at runtime.

The previously-mentioned section on setting up Play Services also covers requesting runtime permissions.

Clients, Connections, and Callbacks

Play Services runs in its own process, one that appears to be continuously monitoring the user's location. In order to get location data from this process, we need to establish some sort of IPC (inter-process communication) with it. The low-level implementation of this is handled by the Play Services Android library project. However, we do need to set some things up ourselves.

Specifically, we need to create and use an instance of `GoogleApiClient`, our gateway to the Play Services SDK. The `AbstractGoogleApiClientActivity` — described in the section on setting up Play Services — handles a lot of this for us. What we need to do is override a few methods in our concrete `WeatherDemo` subclass of `AbstractGoogleApiClientActivity`:

```
package com.commonsware.android.weather2;

import android.Manifest;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;
import com.google.android.gms.common.api.GoogleApiClient;
import com.google.android.gms.location.LocationServices;

public class WeatherDemo extends AbstractGoogleApiClientActivity {
  private static final String[] PERMS=
    {Manifest.permission.ACCESS_FINE_LOCATION};

  @Override
  protected GoogleApiClient.Builder configureApiClientBuilder(
    GoogleApiClient.Builder b) {
    return(b.addApi(LocationServices.API));
  }

  @Override
  protected String[] getDesiredPermissions() {
    return(PERMS);
  }

  @Override
  protected void handlePermissionDenied() {
    Toast
      .makeText(this, R.string.msg_no_perm, Toast.LENGTH_LONG)
      .show();
    finish();
  }
```

```
  @Override
  public void onConnected(Bundle bundle) {
    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
        .add(android.R.id.content,
          new WeatherFragment()).commit();
    }
  }

  @Override
  public void onConnectionSuspended(int i) {
    Log.w(((Object)this).getClass().getSimpleName(),
      "onConnectionSuspended() called, whatever that means");
  }
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/WeatherDemo.java)

Specifically:

- `configureApiClientBuilder()` needs to flesh out the details of what sorts of APIs we are looking to use in this app. In this case, we call `addApi()` on the supplied `GoogleApiClient.Builder`, requesting the `LocationServices.API`, to be able to get at the relevant portion of the Play Services SDK.
- `getDesiredPermissions()` returns an array of the names of the runtime permissions that we need in order to use this API. In this case, we are asking for `ACCESS_FINE_LOCATION` (though, in truth, `ACCESS_COARSE_LOCATION` would suffice for getting weather forecasts). The manifest also has the `<uses-permission>` element for `ACCESS_FINE_LOCATION`.
- `handlePermissionDenied()` will be called if we request the permission and we do not have it when we get control back. This means that either the user denied it now or the user denied it earlier and checked the "Don't ask again" checkbox to stop being bothered about runtime permissions. We could use `shouldShowPermissionRequestRationale()` and perhaps take some steps to educate the user. This is a book example, so we just `finish()` the activity and move along with our day.
- `onConnected()`, from the `GoogleApiClient.ConnectionCallbacks`, will be called if we now have access to the `LocationServices.API` that we requested. At this point, it is safe for us to show a `WeatherFragment` that will use this API to go get the location and, from that, get the weather forecast.
- `onConnectionSuspended()`, also from the `GoogleApiClient.ConnectionCallbacks`, might get called, for uncertain reasons. Here, we are largely ignoring this condition.

The upshot is that, if things go as expected, we will show the `WeatherFragment` when we can get the location and the subsequent forecast.

Finding the Current Location

Given all that setup, actually getting the location is almost anti-climactic.

To find the current location, given a connected `GoogleApiClient`, just call the static `getLastLocation()` method on `LocationServices.FusedLocationApi`, passing in the `GoogleApiClient` instance as a parameter. This usually will return a non-`null` `Location` object, using the same `Location` class that you would use with `LocationManager`.

In the sample, the `run()` method checks to see if `getLastLocation()` returns `null` or not. If the location is `null`, it schedules `run()` to be invoked again in one second, using `postDelayed()` on some suitable `View` (in this case, the `WebView` for displaying the results). If, however, we do have a valid location, `run()` invokes a `FetchForecastTask`, as did the original version of this sample:

```
@SuppressWarnings("MissingPermission")
@Override
public void run() {
  Location location=
    LocationServices.FusedLocationApi.getLastLocation(getPlayServices());

  if (location==null) {
    getListView().postDelayed(this, 1000);
  }
  else {
    fetchForecast(location);
  }
}
```

(from Location/FusedNew/app/src/main/java/com/commonsware/android/weather2/WeatherFragment.java)

The fact that we are using `postDelayed()` here is why we use `removeCallbacks()` in `onPause()`, to stop polling for `getLastLocation()` when we are disconnecting from the `LocationClient`.

Note that the documentation for `getLastLocation()` states "If a location is not available, which should happen very rarely, `null` will be returned." The "very rarely" part indicates that Play Services is constantly checking for the user's location, possibly because location providers are not available.

Requesting Location Updates

As with `LocationManager`, you can use `LocationClient` to be delivered location updates as the device moves, via `requestLocationUpdates()`. There are two major axes of control you have over these updates: the way the locations are delivered to you, and the `LocationRequest` that configures what updates you receive.

Delivery Options

A foreground application would use forms of `requestLocationUpdates()` that take a `LocationListener` as a parameter. Despite the class being named the same, this is a separate implementation of the `LocationListener` interface. The Play Services one (`com.google.android.gms.location.LocationListener`) only requires a single method: `onLocationChanged()`, which is handed a `Location` object representing a location fix.

A background application would use the `requestLocationUpdates()` that takes a `PendingIntent` instead of a `LocationListener`, where that `PendingIntent` can do whatever you wish (start an activity, start a service, send a broadcast). The location itself is delivered in the form of an `Intent` extra, keyed as `KEY_LOCATION_CHANGED`, with a value in the form of a `Location` object.

Request Options

All forms of `requestLocationUpdates()` take a `LocationRequest` object describing what you want in terms of updates. Unlike with `LocationManager`, you do not specify specific location technologies (e.g., GPS). You also lack the fine-grained control of the `Criteria` object (e.g., to require the location to have speed data). However, you do have some measure of control, via various setters on `LocationRequest`.

Frequency

Calling `setInterval()` indicates approximately how frequently you wish to receive location updates. The key word here is "approximately", as you will receive updates more or less frequently than the number of milliseconds you specify as the desired interval. However, your requested interval is taken into account, and the longer of an interval you provide, the less power your app will consume.

To help prevent being flooded with location data, you can also call `setFastestInterval()`, which will throttle the actual updates to be no more frequent than the number of milliseconds that you state.

Priority

`setPriority()` allows you to control the accuracy and power consumption of your app's request, by specifying one of four possible priority levels:

- `PRIORITY_HIGH_ACCURACY` will tend to use GPS and therefore will consume more power
- `PRIORITY_BALANCED_POWER_ACCURACY` will try to consume somewhat less power
- `PRIORITY_LOW_POWER` will try to consume even less power
- `PRIORITY_NO_POWER` indicates that you want to consume no additional power over any other requests, but to get what you can (akin to the "passive provider" available with `LocationManager`)

Duration

You can proactively cancel receiving further updates by calling `removeUpdates()`, passing in your delivery option from `requestLocationUpdates()`:

- The same `LocationListener` as you used to request the updates, or
- An equivalent `PendingIntent` to the one that you used to request the updates

You can also automatically expire your requested updates by one of three means:

- `setNumUpdates()` indicates exactly how many location fixes that you want to receive (e.g., 1) and discontinues the updates after that number
- `setExpirationDuration()` indicates how long you wish to receive updates, expressed as a number of milliseconds from now
- `setExpirationTime()` indicates when you wish to discontinue updates, expressed in the form of the number of milliseconds since the device turned on (e.g., the same time base as is used by `elapsedRealtime()` on the `SystemClock` class)

For example, an improved version of the sample shown in this chapter would use a `LocationRequest` with `setNumUpdates(1)`, instead of the one-second polling of

`getLastLocation()`. In fact, we will see such an improved version in the next section.

I Can Haz Location?

One common complaint among Android developers is that there is no way for developers to enable location providers, like GPS. This is for privacy reasons. Users should be able to control whether apps can track their movements. However, to enable location providers, the user had to go into the Settings app, which was aggravating.

In early 2015, Google added `SettingsApi` to the fused location provider portion of the Play Services SDK. This allows apps to find out if we are capable of using the fused location provider, and if not, pop up a dialog where the user can agree to enable location tracking.

The `Location/FusedPeriodic` sample application demonstrates this, plus the APIs used for periodic location updates. In truth, we will only get a single location fix, as hinted in the previous section, but this example could be extended to update more than once if needed.

The business logic, and some of the code, is the same as in the previous sample: fetch the location, then show a weather forecast for that location. What differs is in how we are fetching the location, as we use `SettingsApi` and `requestLocationUpdates()`.

Defining a Location Request

Core to both finding out whether we can use the fused location provider, and later getting location fixes, will be to define a `LocationRequest` object. Fortunately, this is a pure POJO, without any ties to any `Context` or other existing Play Services SDK objects. Hence, we can declare it as an ordinary data member and initialize it in `onCreate()` of the revised `WeatherFragment`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setRetainInstance(true);

  Retrofit retrofit=
    new Retrofit.Builder()
```

```
        .baseUrl("https://api.weather.gov")
        .addConverterFactory(GsonConverterFactory.create())
        .build();

    nws=retrofit.create(NWSInterface.class);
    request=new LocationRequest()
              .setNumUpdates(1)
              .setExpirationDuration(60000)
              .setInterval(1000)
              .setPriority(LocationRequest.PRIORITY_LOW_POWER);
  }
```

(from Location/FusedPeriodic/app/src/main/java/com/commonsware/android/weather3/WeatherFragment.java)

In `onCreate()`, we indicate that the `LocationRequest`:

- Only needs to provide us with a single location fix (`setNumUpdates(1)`)
- Can give up automatically if we do not get a location fix within the first minute (`setExpirationDuration(60000)`)
- Should start working fairly quickly to get us our fix (`setInterval(1000)`)
- Can optimize for power over accuracy (`setPriority(LocationRequest.PRIORITY_LOW_POWER)`)

The `setInterval()` call may seem odd, given that we are only seeking one fix. Leaving this out, though, means that you *never* get a fix, for unclear reasons.

Also, while we are requesting `PRIORITY_LOW_POWER`, and we do not need a particularly accurate fix just to get a weather forecast, we still request `ACCESS_FINE_LOCATION` in the manifest. Without this, once again we seem to never get a fix.

Another issue comes with the expiration value. `setExpirationDuration()` calculates the expiration time based on when the `LocationRequest` object is *created*, not when it is *used*. That's bad, and we will see where that can bite us a bit later in this chapter.

Requesting and Reacting to Settings Status

In the original example, once we were connected to the Play Services engine and our `WeatherFragment` was created, we would get the last-known location and try to fetch a forecast. If there was no location, we would just ask again every second. This is not a great solution:

- We might never get a location, because the user has disabled location tracking
- We might never get a location, because the environment is unsuitable (e.g., underground parking garage)
- It does not give Play Services much information about what we need in terms of a location fix

Therefore, this sample changes `onViewCreated()` to call a private `requestSettings()` method, so we can find out if location tracking is enabled and, if not, perhaps ask the user to enable it:

```
private void requestSettings() {
  LocationSettingsRequest.Builder b=
      new LocationSettingsRequest.Builder()
            .addLocationRequest(request);
  PendingResult<LocationSettingsResult> result=
      LocationServices.SettingsApi.checkLocationSettings(getPlayServices(),
                                                         b.build());

  result.setResultCallback(this);
}
```

(from Location/FusedPeriodic/app/src/main/java/com/commonsware/android/weather3/WeatherFragment.java)

Here, we create a `LocationSettingsRequest.Builder` and pass it our already-defined `LocationRequest` via `addLocationRequest()`. This way, Play Services will know the sort of location data that we will be looking for later and can tell us whether or not that is presently possible.

We then `build()` the `LocationSettingsRequest` and pass it to `checkLocationSettings()` on the `LocationServices.SettingsApi` class. This returns a `PendingResult`, specifically of a `LocationSettingsResult` type. We call `setResultCallback()` to indicate that the fragment itself should be notified about the results of this request. That is why `WeatherFragment` now implements the `ResultCallback` interface for `LocationSettingsResult`, which in turn requires us to implement an `onResult()` method that takes a `LocationSettingsResult` as a parameter:

```
@Override
public void onResult(LocationSettingsResult result) {
  boolean thingsPlumbBusted=true;

  switch(result.getStatus().getStatusCode()) {
    case LocationSettingsStatusCodes.SUCCESS:
```

```
        requestLocations();
        thingsPlumbBusted=false;
        break;

      case LocationSettingsStatusCodes.RESOLUTION_REQUIRED:
        try {
          result
            .getStatus()
            .startResolutionForResult(getActivity(),
                                      SETTINGS_REQUEST_ID);
          thingsPlumbBusted=false;
        }
        catch (IntentSender.SendIntentException e) {
          // oops
        }
        break;

      case LocationSettingsStatusCodes.SETTINGS_CHANGE_UNAVAILABLE:
        // more oops
        break;
    }

    if (thingsPlumbBusted) {
      Toast
        .makeText(getActivity(),
                  R.string.settings_resolution_fail_msg,
                  Toast.LENGTH_LONG)
        .show();
      getActivity().finish();
    }
  }
```

(from Location/FusedPeriodic/app/src/main/java/com/commonsware/android/weather3/WeatherFragment.java)

What we are hoping for is `LocationSettingsStatusCodes.SUCCESS` as the status code out of the result's `Status` object (obtained via `getStatus()`). This means that our proposed location updates should succeed, and we can go ahead and make that request. We do that via a call to a private `requestLocations()` method that we will explore a bit later.

However, instead we might get `LocationSettingsStatusCodes.RESOLUTION_REQUIRED`. This means that the user has disabled location providers necessary to fulfill the request, but that we could prompt the user to enable location tracking. To do this, we call `startResolutionForResult()` on that `Status` object, passing in our `Activity` along with a locally-unique integer.

This will display the following dialog-themed `Activity`:

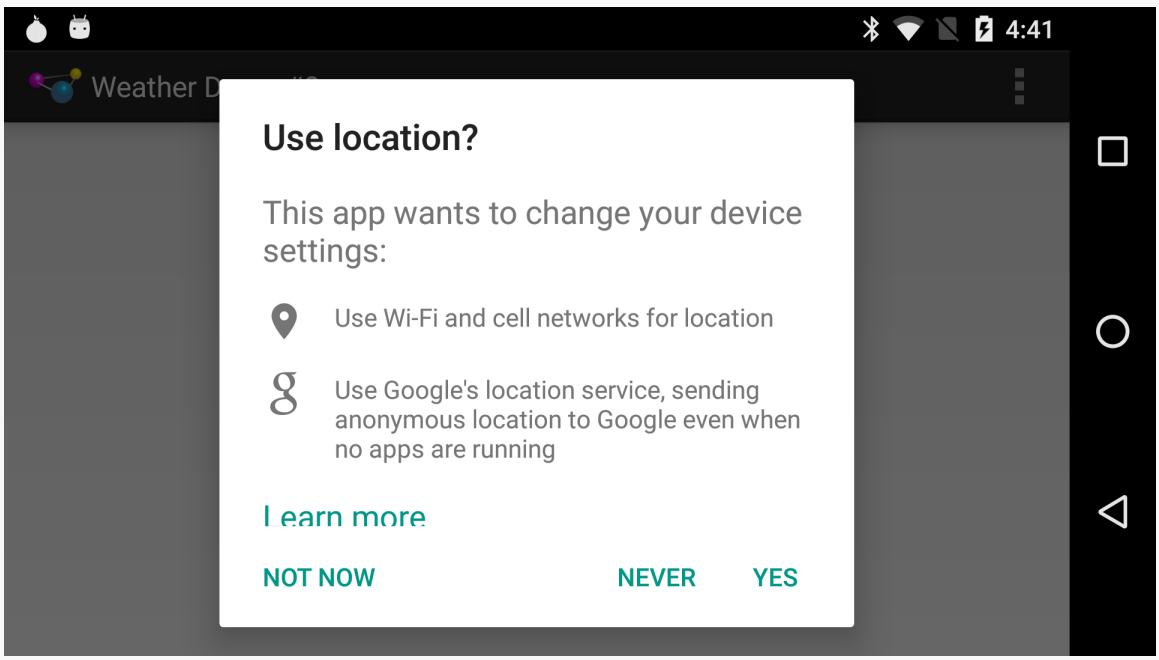


Figure 866: Location Enable Dialog

Google Maps users will recognize it as being akin to the one that appears if you try using certain Maps features (e.g., navigation) and do not have location tracking enabled in Settings.

The user has three choices:

1. Click "Yes"
2. Click "No"
3. Click "Never", which not only has an immediate effect akin to "No" but also will immediately fail all future attempts at resolution from your app, until such time as the user clears the app's data or reinstalls the app from scratch

You can block that "Never" option via a call to `setAlwaysShow(true)` on the `LocationSettingsRequest.Builder` before calling `checkLocationSettings()`.

That should eventually trigger a call to `onActivityResult()` *on the activity* – `startResolutionForResult()` knows nothing about fragments. There are two possibilities here: the user accepts our request to enable location tracking, or the user denies it. If the user accepts our request, `onResult()` will be called again with

`LocationSettingsStatusCodes.SUCCESS`, at which time we can request location updates. If the user rejects the request, we need to arrange to stop asking. By default, if we do nothing, if the user rejects the request, we will be called with `LocationSettingsStatusCodes.RESOLUTION_REQUIRED` again and will pop up the dialog again. That is why, in the `WeatherDemo` activity, we have an `onActivityResult()` to catch when the user completes the dialog triggered by `startResolutionForResult()`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
  if (requestCode==WeatherFragment.SETTINGS_REQUEST_ID) {
    if (resultCode==Activity.RESULT_CANCELED) {
      finish();
    }
    else {
      // this should not be needed, but apparently is in 8.1
      WeatherFragment f=
        (WeatherFragment)getFragmentManager().findFragmentById(android.R.id.content);
      f.requestLocations();
    }
  }
}
```

(from Location/FusedPeriodic/app/src/main/java/com/commonsware/android/weather3/WeatherDemo.java)

The locally-unique integer provided to `startResolutionForResult()`, under the covers, is used for a `startActivityForResult()` call, which is why we get control in `onActivityResult()`. If the request is for this settings dialog, and if the user canceled our request, we `finish()` the activity, as we cannot do anything anymore and may as well close up shop. There are probably other ways of handling this condition that will prevent `onResult()` from getting called again.

If the `resultCode` is `Activity.RESULT_OK`, though, then the user presumably is allowing us to request locations, having clicked "Yes" on the dialog. For some reason, on version 8.1 of the Play Services SDK, this does *not* trigger a fresh call to the `onResult()` method, the way it used to. So, we have to get our `WeatherFragment` and call `requestLocations()` from `onActivityResult()` instead.

Speaking of `onResult()`, it is possible that the status code is neither of those values. In those cases, we are also stuck — Play Services is indicating that we will not be able to get the locations that we are requesting. So, there, we show a `Toast` and `finish()` the activity.

Requesting "Periodic Locations"

The `requestLocations()` method will be triggered once `SettingsApi` gives us "the go-ahead" via the `onResult()` method:

```
@SuppressWarnings("MissingPermission")
void requestLocations() {
  PendingResult<Status> result=
      LocationServices.FusedLocationApi
        .requestLocationUpdates(getPlayServices(), request, this);

  result.setResultCallback(new ResultCallback<Status>() {
    @Override
    public void onResult(Status status) {
      if (status.isSuccess()) {
        Toast
            .makeText(getActivity(),
                R.string.location_req_success_msg,
                Toast.LENGTH_LONG)
            .show();
      } else {
        Toast
            .makeText(getActivity(), status.getStatusMessage(),
                Toast.LENGTH_LONG)
            .show();
        getActivity().finish();
      }
    }
  });
}
```

(from Location/FusedPeriodic/app/src/main/java/com/commonsware/android/weather3/WeatherFragment.java)

Here, we start off by calling `requestLocationUpdates()`, passing in the `LocationRequest` that we created before, along with our `LocationListener` implementation, which happens to be the fragment itself.

We can, if we want, attach another `ResultCallback` object to the `PendingResult` returned by `requestLocationUpdates()`. This way, we can find out if our request for location updates was successfully queued or not. Here, we do that, using an instance of an anonymous inner class of `PendingResult`. We show a `Toast` regardless of success or failure; we also `finish()` the method on failure.

What should happen, if the request was successful, is that we will get one fix delivered to `onLocationChanged()` on our `LocationListener`. There, we use Retrofit as before to retrieve the forecast and update the `ListView`:

```
@Override
public void onLocationChanged(Location location) {
  double roundedLat=(double)Math.round(location.getLatitude()*10000d)/10000d;
  double roundedLon=(double)Math.round(location.getLongitude()*10000d)/10000d;

  nws.getForecast(roundedLat, roundedLon)
    .enqueue(new Callback<WeatherResponse>() {
      @Override
      public void onResponse(Call<WeatherResponse> call,
                             Response<WeatherResponse> response) {
        if (response.code()==200) {
          adapter=new ForecastAdapter(response.body().properties.periods);
          setListAdapter(adapter);
        }
        else {
          Toast.makeText(getActivity(), R.string.msg_nws,
            Toast.LENGTH_LONG).show();
        }
      }

      @Override
      public void onFailure(Call<WeatherResponse> call, Throwable t) {
        Toast.makeText(getActivity(), t.getMessage(),
          Toast.LENGTH_LONG).show();
        Log.e(getClass().getSimpleName(),
          "Exception from Retrofit request to National Weather Service", t);
      }
    });
}
```

(from Location/FusedPeriodic/app/src/main/java/com/commonsware/android/weather3/WeatherFragment.java)

If, after a minute of trying, we do not get the location fix, Play Services will stop trying, based on our `setExpirationDuration()` value on the `LocationRequest` object.

However, this is where we run into a problem with using that `LocationRequest` created originally. It appears that the `LocationRequest` calculates the time to give up based on when the `LocationRequest` is created. It is not based on the time when we call `requestLocationUpdates()`. In many cases, there will be little delay between those two points in time, and so the different is negligible. But if the user gets the enable-location dialog and leaves it open for a minute, when we call `requestLocationUpdates()`, *we are already expired*. However, we do not find out about this, and we just never get a location fix. Hence, rather than creating a single instance of `LocationRequest`, have a `buildLocationRequest()` method that can return the instance to you, newly created, so you have the full expiration time to work with.

Also, in `onPause()`, we call `removeLocationUpdates()`, so that we minimize power drain while we are not in the foreground.

This sample app is less-than-optimal in its handling of configuration changes. Everything works, but we wind up re-requesting location updates on each configuration change. A better implementation would note if we already have our location fix and therefore no longer need to request location updates.

Working with the Clipboard

Being able to copy and paste is something that mobile device users seem to want almost as much as their desktop brethren. Most of the time, we think of this as copying and pasting text, but one could copy and paste other things, such as `Uri` values pointing to more elaborate forms of content.

In this chapter, we will explore how to work with the modern clipboard APIs. Here, "modern" refers to `android.content.ClipboardManager`. Android 1.x and 2.x used `android.text.ClipboardManager`, which still exists in the Android SDK for backwards-compatibility reasons. However, most modern development should use `android.content.ClipboardManager`.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Working with the Clipboard

`ClipboardManager` can be obtained via a call to `getSystemService()` on any handy `Context`.

The old Android 1.x/2.x API was dominated by three methods, all focused on plain text:

- `setText()`, to put text on the clipboard
- `hasText()`, to indicate if the clipboard has something on it
- `getText()`, to retrieve the text on the clipboard

Those methods still exist, but they have been deprecated as of API Level 11.

Their replacements are:

- `setPrimaryClip()`, to put something on the clipboard
- `hasPrimaryClip()`, to indicate if the clipboard has something on it
- `getPrimaryClip()`, to retrieve something from the clipboard

Here, the "something" winds up being in the form of `ClipData` objects, which can hold:

1. plain text
2. a `Uri` (e.g., to a piece of music)
3. an `Intent`

The `Uri` means that you can put anything on the clipboard that can be referenced by a `Uri`... and if there is nothing in Android that lets you reference some data via a `Uri`, you can invent your own content provider to handle that chore for you. Furthermore, a single `ClipData` can actually hold as many of these as you want, each represented as individual `ClipData.Item` objects. As such, the possibilities are endless.

There are static factory methods on `ClipData`, such as `newUri()`, that you can use to create your `ClipData` objects. In fact, that is what we use in the SystemServices/ClipMusic sample project and the `MusicClipper` activity.

`MusicClipper` has the classic two-big-button layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="Pick"
    android:onClick="pickMusic"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="Play"
    android:onClick="playMusic"
  />
</LinearLayout>
```

(from SystemServices/ClipMusic/app/src/main/res/layout-land/main.xml)

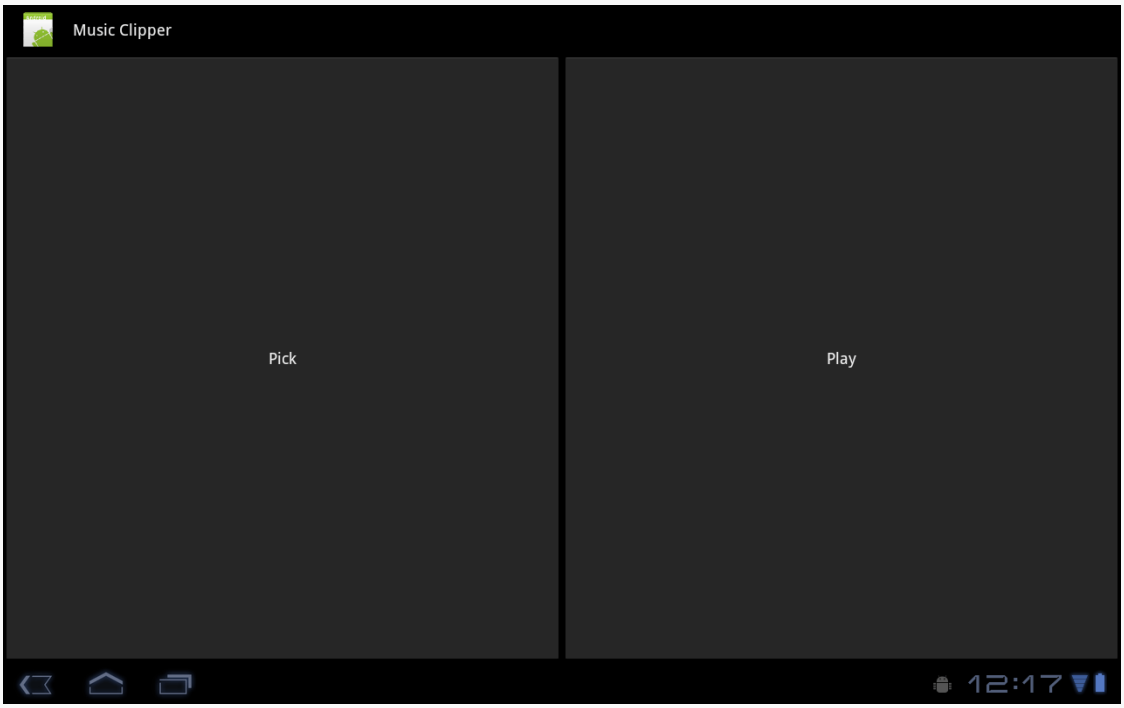


Figure 867: The Music Clipper main screen

In `onCreate()`, we get our hands on our `ClipboardManager` system service:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  clipboard=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
}
```

(from SystemServices/ClipMusic/app/src/main/java/com/commonsware/android/clip/music/MusicClipper.java)

Tapping the "Pick" button will let you pick a piece of music, courtesy of the `pickMusic()` method wired to that `Button` object:

```
public void pickMusic(View v) {
  Intent i=new Intent(Intent.ACTION_GET_CONTENT);

  i.setType("audio/*");
  startActivityForResult(i, PICK_REQUEST);
}
```

(from SystemServices/ClipMusic/app/src/main/java/com/commonsware/android/clip/music/MusicClipper.java)

Here, we tell Android to let us pick a piece of music from any available audio MIME type (`audio/*`). Fortunately, Android has an activity that lets us do that:

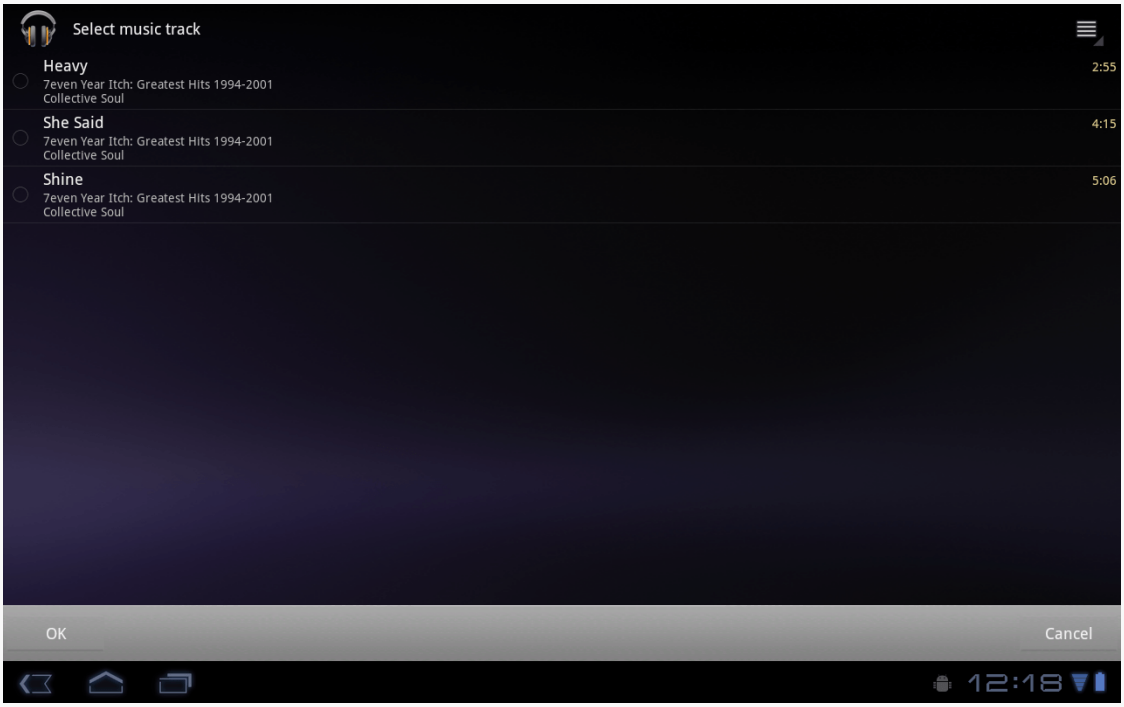


Figure 868: The XOOM tablet's music track picker

We get the result in `onActivityResult()`, since we used `startActivityForResult()` to pick the music. There, we package up the `content://` `Uri` to the music into a `ClipData` object and put it on the clipboard:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode == PICK_REQUEST) {
    if (resultCode == RESULT_OK) {
      ClipData clip=
```

```
          ClipData.newUri(getContentResolver(), "Some music",
                          data.getData());

      try {
        clipboard.setPrimaryClip(clip);
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(), "Exception clipping Uri", e);
        Toast.makeText(this, "Exception: " + e.getMessage(),
                       Toast.LENGTH_SHORT).show();
      }
    }
  }
}
```

(from SystemServices/ClipMusic/app/src/main/java/com/commonsware/android/clip/music/MusicClipper.java)

Note that there is a significant bug in Android 4.3 that, until it is fixed, will require you to do a bit more error-handling with your clipboard operations. That is why we have our `setPrimaryClip()` call wrapped in a `try/catch` blog, even though `setPrimaryClip()` does not throw a checked exception. The rationale for this will be discussed later in this chapter.

The catch with rich data on the clipboard is that somebody has to know about the sort of information you are placing on the clipboard. Eventually, the Android development community will work out common practices in this area. Right now, though, you can certainly use it within your own application (e.g., clipping a note and pasting it into another folder).

Since putting `ClipData` onto the clipboard involves a call to `setPrimaryClip()`, it should not be surprising that the reverse operation — getting a `ClipData` from the clipboard — uses `getPrimaryClip()`. However, since you do not know where this clip came from, you need to validate that it has what you expect and to let the user know when the clipboard contents are not something you can leverage.

The "Play" button in our UI is wired to a `playMusic()` method. This will only work when we have pasted a `Uri ClipData` to the clipboard pointing to a piece of music. Since we cannot be sure that the user has done that, we have to sniff around:

```
public void playMusic(View v) {
  ClipData clip=clipboard.getPrimaryClip();

  if (clip == null) {
    Toast.makeText(this, "There is no clip!", Toast.LENGTH_LONG)
```

```
          .show();
    }
    else {
      ClipData.Item item=clip.getItemAt(0);
      Uri song=item.getUri();

      if (song != null
          && getContentResolver().getType(song).startsWith("audio/")) {
        startActivity(new Intent(Intent.ACTION_VIEW, song));
      }
      else {
        Toast.makeText(this, "There is no song!", Toast.LENGTH_LONG)
             .show();
      }
    }
  }
```

(from SystemServices/ClipMusic/app/src/main/java/com/commonsware/android/clip/music/MusicClipper.java)

First, there may be nothing on the clipboard, in which case the `ClipData` returned by `getPrimaryClip()` would be `null`. Or, there may be stuff on the clipboard, but it may not have a `Uri` associated with it (`getUri()` on `ClipData`). Even then, the `Uri` may point to something other than music, so even if we get a `Uri`, we need to use a `ContentResolver` to check the MIME type (`getContentResolver().getType()`) and make sure it seems like it is music (e.g., starts with `audio/`). Then, and only then, does it make sense to try to start an `ACTION_VIEW` activity on that `Uri` and hope that something useful happens. Assuming you clipped a piece of music with the "Pick" button, "Play" will kick off playback of that song.

ClipData and Drag-and-Drop

API Level 11 also introduced Android's first built-in drag-and-drop framework. One might expect that this would be related entirely to `View` and `ViewGroup` objects and have nothing to do with the clipboard. In reality, the drag-and-drop framework leverages `ClipData` to say what it is that is being dragged and dropped. You call `startDrag()` on a `View`, supplying a `ClipData` object, along with some objects to help render the "shadow" that is the visual representation of this drag operation. A `View` that can receive objects "dropped" via drag-and-drop needs to register an `OnDragListener` to receive drag events as the user slides the shadow over the top of the `View` in question. If the user lifts their finger, thereby dropping the shadow, the recipient `View` will get an `ACTION_DROP` drag event, and can get the `ClipData` out of the event.

The chapter on drag-and-drop goes into this in much greater detail.

Monitoring the Clipboard

API Level 11 added the capability for an app to monitor what is put on the clipboard, including things put on the clipboard by other apps.

This is a somewhat esoteric feature, but one that perhaps has some valid use cases. Mostly, it would be used by something *not* in the foreground, since the foreground activity is probably what is adding material to the clipboard. A service, or perhaps an activity that has moved to the background, could use this feature to find out about new clipboard entries.

To monitor the clipboard, you simply call `addPrimaryClipChangedListener()` on `ClipboardMonitor`, passing an implementation of an `OnPrimaryClipChangedListener` interface. That object, in turn, will be called with `onPrimaryClipChanged()` whenever there is a new clipboard entry. Later on, you can call `removePrimaryClipChangedListener()` to stop being notified about new clipboard entries.

For example, here is `MainActivity` from the SystemServices/ClipboardMonitor sample project:

```
package com.commonsware.android.clipmon;

import android.app.Activity;
import android.content.ClipboardManager;
import android.content.ClipboardManager.OnPrimaryClipChangedListener;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends Activity implements
    OnPrimaryClipChangedListener {
  private ClipboardManager cm=null;
  private TextView lastClip=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    lastClip=(TextView)findViewById(R.id.last_clip);
    cm=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
```

```
  }

  @Override
  public void onStart() {
    super.onStart();
    cm.addPrimaryClipChangedListener(this);
  }

  @Override
  public void onStop() {
    cm.removePrimaryClipChangedListener(this);
    super.onStop();
  }

  @Override
  public void onPrimaryClipChanged() {
    lastClip.setText(cm.getPrimaryClip().getItemAt(0)
                       .coerceToText(this));
  }
}
```

(from SystemServices/ClipboardMonitor/app/src/main/java/com/commonsware/android/clipmon/MainActivity.java)

Here, we:

- Retrieve the `ClipboardManager` in `onCreate()`
- Register for clipboard events via `addPrimaryClipChangedListener()` in `onStart()`
- Unregister from clipboard events via `removePrimaryClipChangedListener()` in `onStop()`
- Convert the first item (`getItemAt(0)`) of the primary clip (`getPrimaryClip()`) to text (`coerceToText(this)`), and stuff the results into a `TextView`

In theory, this activity will display new clipboard entries as they arrive. In practice, it will only do so while it is in the foreground, and so it would require something in the background to add something to the clipboard. That is not a particularly useful example... except to test the bug outlined in the next section.

The Android 4.3 Clipboard Bug

AndroidPolice reported on a fairly unpleasant bug in Android 4.3. While this bug was fixed in Android 4.4, there is little evidence that Google will be releasing a fix for

Android 4.3 devices, which means that this problem will plague developers into 2015 and perhaps beyond.

The bug stems from the clipboard monitoring facility. If an app has used `addPrimaryClipChangedListener()`, *any other app* that tries to paste to the clipboard will crash.

The first crash will be a `SecurityException`:

```
java.lang.SecurityException: uid ... does not have
android.permission.UPDATE_APP_OPS_STATS
```

The second and subsequent times this occurs on the device, it will be an `IllegalStateException`:

```
java.lang.IllegalStateException: beginBroadcast() called while already
in a broadcast
```

The only resolution is to unregister the clipboard listener... and hope that the first crash has not occurred. If it has, a **full reboot of the device is required** to fix the broken system.

If Your App Monitors the Clipboard…

If you have a component, such as a long-running service, that is monitoring the clipboard, please ensure that the users have an easy way to stop that behavior, even if it means stopping your whole service. While this may mean that your app has seriously degraded functionality, the alternative is that the user has to keep rebooting their device while your app is installed.

If Your App Pastes to the Clipboard…

If you are pasting to the clipboard, with `setPrimaryClip()` or the older `setText()`, you will want to throw a `try/catch` block around those calls, so you catch the `RuntimeExceptions` that will be thrown.

However, you will need to tell your users that they are now fairly well screwed, needing to both find the clipboard-monitoring app and learn how to control it (or

uninstall/disable it, if needed), plus reboot their device, in order to paste to the clipboard again.

Telephony

Many, if not most, Android devices will be phones. As such, not only will users be expecting to place and receive calls using Android, but you will have the opportunity to help them place calls, if you wish.

Why might you want to?

1. Maybe you are writing an Android interface to a sales management application (a la Salesforce.com) and you want to offer users the ability to call prospects with a single button click, and without them having to keep those contacts both in your application and in the phone's contacts application
2. Maybe you are writing a social networking application, and the roster of phone numbers that you can access shifts constantly, so rather than try to "sync" the social network contacts with the phone's contact database, you let people place calls directly from your application
3. Maybe you are creating an alternative interface to the existing contacts system, perhaps for users with reduced motor control (e.g., the elderly), sporting big buttons and the like to make it easier for them to place calls

Whatever the reason, Android has the means to let you manipulate the phone just like any other piece of the Android system.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on working with multiple activities.

Report To The Manager

To get at much of the phone API, you use the `TelephonyManager`. That class lets you do things like:

1. Determine if the phone is in use via `getCallState()`, with return values of `CALL_STATE_IDLE` (phone not in use), `CALL_STATE_RINGING` (call requested but still being connected), and `CALL_STATE_OFFHOOK` (call in progress)
2. Find out the SIM ID (IMSI) via `getSubscriberId()`
3. Find out the phone type (e.g., GSM) via `getPhoneType()` or find out the data connection type (e.g., GPRS, EDGE) via `getNetworkType()`

You Make the Call!

You can also initiate a call from your application, such as from a phone number you obtained through your own Web service. To do this, simply craft an `ACTION_DIAL` `Intent` with a `Uri` of the form `tel:NNNNN` (where `NNNNN` is the phone number to dial) and use that `Intent` with `startActivity()`. This will not actually dial the phone; rather, it activates the dialer activity, from which the user can then press a button to place the call.

For example, let's look at the Phone/Dialer sample application. Here's the crude-but-effective layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    >
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Number to dial:"
      />
    <EditText android:id="@+id/number"
      android:layout_width="match_parent"
```

```
      android:layout_height="wrap_content"
      android:cursorVisible="true"
      android:editable="true"
      android:singleLine="true"
    />
  </LinearLayout>
  <Button android:id="@+id/dial"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Dial It!"
    android:onClick="dial"
  />
</LinearLayout>
```

(from Phone/Dialer/app/src/main/res/layout/main.xml)

We have a labeled field for typing in a phone number, plus a button for dialing said number.

The Java code simply launches the dialer using the phone number from the field:

```
package com.commonsware.android.dialer;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class DialerDemo extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
  }

  public void dial(View v) {
    EditText number=(EditText)findViewById(R.id.number);
    String toDial="tel:"+number.getText().toString();

    startActivity(new Intent(Intent.ACTION_DIAL, Uri.parse(toDial)));
  }
}
```

(from Phone/Dialer/app/src/main/java/com/commonsware/android/dialer/DialerDemo.java)

The activity's own UI is not that impressive:

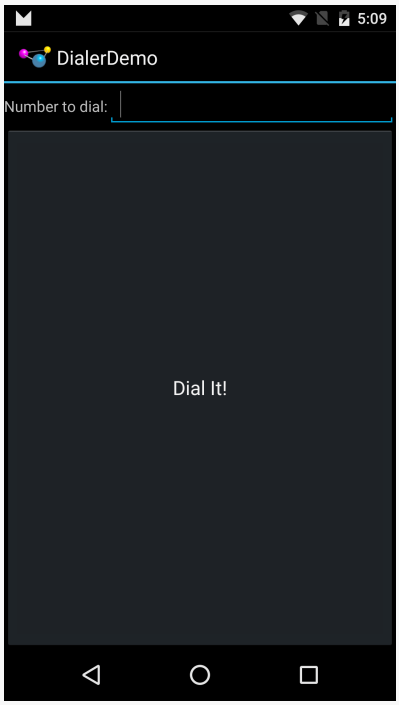


Figure 869: The DialerDemo sample application, as initially launched

However, the dialer you get from clicking the dial button is better, showing you the number you are about to dial:

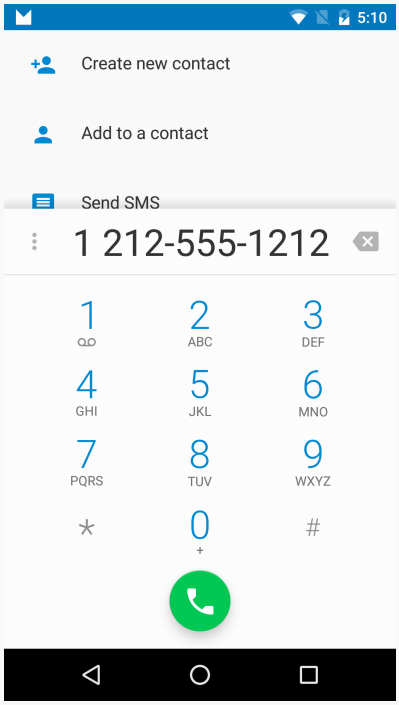


Figure 870: The Android Dialer activity, as launched from DialerDemo

No, Really, You Make the Call!

The good news is that `ACTION_DIAL` works without any special permissions. The bad news is that it only takes the user to the Dialer – the user still has to take action (pressing the green call button) to actually place the phone call.

An alternative approach is to use `ACTION_CALL` instead of `ACTION_DIAL`. Calling `startActivity()` on an `ACTION_CALL` `Intent` will immediately place the phone call, without any other UI steps required. However, you need the `CALL_PHONE` permission in order to use `ACTION_CALL`.

Working With SMS

> Oh, what a tangled web we weave
>
> When first we practice to work with SMS on Android, Eve

(with apologies to Sir Walter Scott)

Android devices have had SMS capability since Android 1.0. However, from a programming standpoint, for years, SMS and Android were intensely frustrating. When the Android SDK was developed, some aspects of working with SMS were put into the SDK, while others were held back. This, of course, did not stop many an intrepid developer from working with the undocumented, unsupported SMS APIs, with varying degrees of success.

After much wailing and gnashing of teeth by developers, Google finally formalized a more complete SMS API in Android 4.4. However, this too has its issues, where some apps that worked fine with the undocumented API will now fail outright, in irreparable fashion, on Android 4.4+.

This chapter starts with the one thing you can do reasonably reliably across Android device versions – send an SMS, either directly or by invoking the user's choice of SMS client. The chapter then examines how to monitor or receive SMS messages (both pre-4.4 and 4.4+) and the SMS-related `ContentProvider` (both pre-4.4 and 4.4+).

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on broadcast `Intents`. One of the samples uses the

ContactsContract provider, so reading that chapter will help you understand that particular sample.

Sending Out an SOS, Give or Take a Letter

While much of Android's SMS capabilities are not in the SDK, sending an SMS is. You have two major choices for doing this:

- Invoke the user's choice of SMS client application, so they can compose a message, track its progress, and so forth using that tool
- Send the SMS directly yourself, bypassing any existing client

Which of these is best for you depends on what your desired user experience is. If you are composing the message totally within your application, you may want to just send it. However, as we will see, that comes at a price: an extra permission.

Sending Via the SMS Client

Sending an SMS via the user's choice of SMS client is very similar to the use of ACTION_SEND described elsewhere in this book. You craft an appropriate Intent, then call startActivity() on that Intent to bring up an SMS client (or allow the user to choose between clients).

The Intent differs a bit from the ACTION_SEND example:

1. You use ACTION_SENDTO, rather than ACTION_SEND
2. Your Uri needs to begin with smsto:, followed by the mobile number you want to send the message to
3. Your text message goes in an sms_body extra on the Intent

For example, here is a snippet of code from the SMS/Sender sample project:

```
    Intent sms=new Intent(Intent.ACTION_SENDTO,
                          Uri.parse("smsto:"+c.getString(2)));

    sms.putExtra("sms_body", msg.getText().toString());

    startActivity(sms);
```

(from SMS/Sender/app/src/main/java/com/commonsware/android/sms/sender/Sender.java)

Here, our phone number is coming out of the third column of a `Cursor`, and the text message is coming from an `EditText` — more on how this works later in this section, when we review the `Sender` sample more closely.

Sending SMS Directly

If you wish to bypass the UI and send an SMS directly, you can do so through the `SmsManager` class, in the `android.telephony` package. Unlike most Android classes ending in `Manager`, you obtain an `SmsManager` via a static `getDefault()` method on the `SmsManager` class. You can then call `sendTextMessage()`, supplying:

1. The phone number to send the text message to
2. The "service center" address — leave this `null` unless you know what you are doing
3. The actual text message
4. A pair of `PendingIntent` objects to be executed when the SMS has been sent and delivered, respectively

If you are concerned that your message may be too long, use `divideMessage()` on `SmsManager` to take your message and split it into individual pieces. Then, you can use `sendMultipartTextMessage()` to send the entire `ArrayList` of message pieces.

For this to work, your application needs to hold the `SEND_SMS` permission, via a child element of your `<manifest>` element in your `AndroidManifest.xml` file.

For example, here is code from `Sender` that uses `SmsManager` to send the same message that the previous section sent via the user's choice of SMS client:

```
    SmsManager
      .getDefault()
      .sendTextMessage(c.getString(2), null,
                       msg.getText().toString(),
                       null, null);
```

(from SMS/Sender/app/src/main/java/com/commonsware/android/sms/sender/Sender.java)

Inside the Sender Sample

The `Sender` example application is fairly straightforward, given the aforementioned techniques.

The manifest has both the `SEND_SMS` and `READ_CONTACTS` permissions, because we want to allow the user to pick a mobile phone number from their list of contacts, rather than type one in by hand:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.sender"
  android:installLocation="preferExternal"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="Sender"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from SMS/Sender/app/src/main/AndroidManifest.xml)

If you noticed the `android:installLocation` attribute in the root element, that is to allow this application to be installed onto external storage, such as an SD card.

The layout has a `Spinner` (for a drop-down of available mobile phone numbers), a pair of `RadioButton` widgets (to indicate which way to send the message), an `EditText` (for the text message), and a "Send" `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
>
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <RadioGroup android:id="@+id/means"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    >
      <RadioButton android:id="@+id/client"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="Via Client" />
      <RadioButton android:id="@+id/direct"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Direct" />
  </RadioGroup>
  <EditText
    android:id="@+id/msg"
    android:layout_width="match_parent"
    android:layout_height="0px"
    android:layout_weight="1"
    android:singleLine="false"
    android:gravity="top|left"
  />
  <Button
    android:id="@+id/send"
    android:text="Send!"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="sendTheMessage"
  />
</LinearLayout>
```

(from SMS/Sender/app/src/main/res/layout/main.xml)

Sender uses the same technique for obtaining mobile phone numbers from our contacts as is seen in the chapter on contacts. To support Android 1.x and Android 2.x devices, we implement an abstract class and two concrete implementations, one for the old API and one for the new. The abstract class then has a static method to get at an instance suitable for the device the code is running on:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.os.Build;
import android.widget.SpinnerAdapter;

abstract class ContactsAdapterBridge {
  abstract SpinnerAdapter buildPhonesAdapter(Activity a);

  public static final ContactsAdapterBridge INSTANCE=buildBridge();

  private static ContactsAdapterBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<5) {
      return(new OldContactsAdapterBridge());
    }

    return(new NewContactsAdapterBridge());
  }
}
```

(from SMS/Sender/app/src/main/java/com/commonsware/android/sms/sender/ContactsAdapterBridge.java)

The Android 2.x edition uses ContactsContract to find just the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.widget.SpinnerAdapter;
import android.widget.SimpleCursorAdapter;

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] { Contacts._ID,
                                       Contacts.DISPLAY_NAME,
```

```
                                          Phone.NUMBER
                                        };
    String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Phone.CONTENT_URI,
                            PROJECTION, Phone.TYPE+"=?",
                            ARGS, Contacts.DISPLAY_NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

(from SMS/Sender/app/src/main/java/com/commonsware/android/sms/sender/NewContactsAdapterBridge.java)

... while the Android 1.x edition uses the older `Contacts` provider to find the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;
import android.widget.SimpleCursorAdapter;
import android.widget.SpinnerAdapter;

@SuppressWarnings("deprecation")
class OldContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                        Contacts.Phones.NAME,
                                        Contacts.Phones.NUMBER
                                      };
    String[] ARGS={String.valueOf(Contacts.Phones.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Contacts.Phones.CONTENT_URI,
                            PROJECTION,
                            Contacts.Phones.TYPE+"=?", ARGS,
```

```
                            Contacts.Phones.NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.Phones.NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

(from SMS/Sender/app/src/main/java/com/commonsware/android/sms/sender/OldContactsAdapterBridge.java)

For more details on how those providers work, please see the chapter on contacts.

The activity then loads up the `Spinner` with the appropriate list of contacts. When the user taps the Send button, the `sendTheMessage()` method is invoked (courtesy of the `android:onClick` attribute in the layout). That method looks at the radio buttons, sees which one is selected, and routes the text message accordingly:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.Spinner;

public class Sender extends Activity {
  Spinner contacts=null;
  RadioGroup means=null;
  EditText msg=null;

  @Override
```

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    contacts=(Spinner)findViewById(R.id.spinner);

    contacts.setAdapter(ContactsAdapterBridge
                        .INSTANCE
                        .buildPhonesAdapter(this));

    means=(RadioGroup)findViewById(R.id.means);
    msg=(EditText)findViewById(R.id.msg);
  }

  public void sendTheMessage(View v) {
    Cursor c=(Cursor)contacts.getSelectedItem();

    if (means.getCheckedRadioButtonId()==R.id.client) {
      Intent sms=new Intent(Intent.ACTION_SENDTO,
                            Uri.parse("smsto:"+c.getString(2)));

      sms.putExtra("sms_body", msg.getText().toString());

      startActivity(sms);
    }
    else {
      SmsManager
        .getDefault()
        .sendTextMessage(c.getString(2), null,
                         msg.getText().toString(),
                         null, null);
    }
  }
}
```

(from SMS/Sender/app/src/main/java/com/commonsware/android/sms/sender/Sender.java)

SMS Sending Limitations

Apps running on Android 1.x and 2.x devices are limited to sending 100 SMS messages an hour, before the user starts getting prompted with each SMS message request to confirm that they do indeed wish to send it.

Apps running on Android 4.x devices, the limits are now 30 SMS messages in 30 minutes, according to some source code analysis by Al Sutton.

Monitoring and Receiving SMS

For the purposes of this section, "monitoring" refers to the ability to inspect incoming SMS messages, including reading their contents. In contrast, "receiving" SMS messages is actually consuming the message and storing it somewhere for the user to use.

As it turns out, "monitoring" and "receiving" are much the same thing prior to Android 4.4, but are significantly different in the new API made available in Android 4.4

The Undocumented, Unsupported, Pre-Android 4.4 Way

It is possible for an application to monitor or receive an incoming SMS message... if you are willing to listen on the undocumented `android.provider.Telephony.SMS_RECEIVED` broadcast `Intent`. That is sent by Android whenever an SMS arrives, and it is up to an application to implement a `BroadcastReceiver` to respond to that `Intent` and do something with the message. The Android open source project has such an application — Messaging — and device manufacturers can replace it with something else.

Note that to listen for this broadcast, your app must hold the `RECEIVE_SMS` permission.

The `BroadcastReceiver` can then turn around and use the `SmsMessage` class, in the `android.telephony` package, to get at the message itself, through the following undocumented recipe:

1. Given the received `Intent` (`intent`), call `intent.getExtras().get("pdus")` to get an `Object` array representing the raw portions of the message
2. For each of those "pdus" objects, call `SmsMessage.createFromPdu()` to convert the `Object` into an `SmsMessage` — though to make this work, you need to cast the `Object` to a `byte` array as part of passing it to the `createFromPdu()` static method

The resulting `SmsMessage` object gets you access to the text of the message, the sending phone number, etc.

The SMS_RECEIVED broadcast Intent is broadcast a bit differently than most others in Android. It is an "ordered broadcast", meaning the Intent will be delivered to one BroadcastReceiver at a time. This has two impacts of note:

- In your receiver's <intent-filter> element, you can have an android:priority attribute. Higher priority values get access to the broadcast Intent earlier than will lower priority values. The standard Messaging application has the default priority (undocumented, appears to be 0 or 1), so you can arrange to get access to the SMS before the application does.
- Your BroadcastReceiver can call abortBroadcast() on itself to prevent the Intent from being broadcast to other receivers of lower priority. In effect, this causes your receiver to consume the SMS — the Messaging application will not receive it. So, aborting the broadcast means that your app chose to "receive" the SMS; not aborting the broadcast means that your app is merely "monitoring" the SMS messages that come in.

However, just because the Messaging application has the default priority does not mean all SMS clients will, and so you cannot reliably intercept SMS messages this way. That, plus the undocumented nature of all of this, means that applications you write to receive SMS messages are likely to be fragile in production, breaking on various devices due to device manufacturer-installed apps, third-party apps, or changes to Android itself... such as the changes that came about in Android 4.4.

The Android 4.4+ Way: Monitoring SMS

The code described above still works on Android 4.4, though the formerly-hidden android.provider.Telephony class is now part of the SDK.

The biggest difference, though, is that even if you call abortBroadcast(), the user's chosen SMS messaging client will still receive the message. It is not possible for an app listening for SMS_RECEIVED broadcasts to prevent the user's chosen SMS messaging client from receiving those same messages. This is a substantial change, one that will break or make obsolete many Android applications.

Regardless, if monitoring SMS fits your needs, SMS_RECEIVED can do it.

So, for example, the SMS/Monitor sample project implements a BroadcastReceiver for SMS_RECEIVED, one with slightly elevated priority:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.commonsware.android.sms.monitor" android:versionCode="1" android:versionName="1.0">

  <supports-screens android:largeScreens="true" android:normalScreens="true" android:smallScreens="false"/>

  <uses-permission android:name="android.permission.RECEIVE_SMS"/>

  <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="19"/>

  <application android:icon="@drawable/ic_launcher" android:label="@string/app_name">
    <receiver android:name="Monitor" android:permission="android.permission.BROADCAST_SMS">
      <intent-filter android:priority="2">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
      </intent-filter>
    </receiver>

    <activity android:name="BootstrapActivity" android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from SMS/Monitor/app/src/main/AndroidManifest.xml)

You will notice that the `BroadcastReceiver` not only has the slightly-elevated priority (`android:priority="2"`), but also a required permission (`android:permission="android.permission.BROADCAST_SMS"`). Only apps that hold this permission can send this broadcast in a way that will be picked up by the receiver. Since this permission can only be held by the device firmware, you are protected from "spoof" SMS messages from rogue apps on the device, sending the `SMS_RECEIVED` themselves.

The app also has a do-nothing activity, solely there to activate the manifest-registered `BroadcastReceiver`, which will not work until some component of the app is manually started.

The bulk of the business logic — what little there is of it — lies in the `Monitor` class that is the `BroadcastReceiver`:

```
package com.commonsware.android.sms.monitor;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.telephony.SmsMessage;
```

```
import android.util.Log;

public class Monitor extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Object[] rawMsgs=(Object[])intent.getExtras().get("pdus");

    for (Object raw : rawMsgs) {
      SmsMessage msg=SmsMessage.createFromPdu((byte[])raw);

      if (msg.getMessageBody().toUpperCase().contains("SEKRIT")) {
        Log.w("SMS:"+msg.getOriginatingAddress(),
              msg.getMessageBody());

        abortBroadcast();
      }
    }
  }
}
```

(from SMS/Monitor/app/src/main/java/com/commonsware/android/sms/monitor/Monitor.java)

Here, we retrieve the raw messages from the `Intent` extra, iterate over them, and convert each to an `SmsMessage`. Those that have the magic word in their message body will result in the message being dumped to Logcat, plus the broadcast is aborted. On Android 4.3 and below, this will prevent lower-priority receivers from receiving the SMS. On Android 4.4, the abort request is ignored.

The Android 4.4+ Way: Receiving SMS

Receiving SMS messages, on Android 4.4+, means that you are implementing an SMS client application, one the user might be willing to set as their default SMS client application in Settings. There are other sorts of apps that may temporarily want to be the default SMS client, such as a backup/restore utility, as only the default SMS client will be able to work with the SMS `ContentProvider` suite, such as the inbox.

Receiving the Broadcasts

The default SMS client should be able to handle both SMS and MMS. This is a problem, as while supporting SMS is poorly documented, supporting MMS has almost no documentation whatsoever. However, unless the default SMS client handles MMS, nobody else can (at least, while saving MMS details to the `ContentProvider` suite.

Hence, Google is expecting you to have two `BroadcastReceivers` registered in the manifest: one for SMS and one for MMS. Unfortunately, these cannot readily be combined into a single receiver, because each has its own permission requirement:

- the SMS receiver should require senders to hold `BROADCAST_SMS`
- the MMS receiver should require senders to hold `BROADCAST_WAP_PUSH`

In practice, probably both are held by the OS component that is sending these broadcasts in response to incoming messages of either type. In principle, though, they could be separate, and an individual `<receiver>` can only specify one such permission.

The Android documentation illustrates the `<receiver>` elements that Google expects your SMS client application to have:

```
<!-- BroadcastReceiver that listens for incoming SMS messages -->
<receiver android:name=".SmsReceiver"
        android:permission="android.permission.BROADCAST_SMS">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_DELIVER" />
    </intent-filter>
</receiver>

<!-- BroadcastReceiver that listens for incoming MMS messages -->
<receiver android:name=".MmsReceiver"
    android:permission="android.permission.BROADCAST_WAP_PUSH">
    <intent-filter>
        <action android:name="android.provider.Telephony.WAP_PUSH_DELIVER" />
        <data android:mimeType="application/vnd.wap.mms-message" />
    </intent-filter>
</receiver>
```

Notice that the MMS receiver has both an `<action>` and a `<data>` element in its `<intent-filter>`, which is rather unusual.

On the SMS side, the `Intent` you receive should be the same as the `Intent` you would receive for the `SMS_RECEIVED` broadcast, where you can decode the message(s) and deal with them as you see fit. On the MMS side... there is little documentation.

Other Expectations

Google expects the default SMS client to be able to handle `ACTION_SEND` and `ACTION_SENDTO` for relevant schemes:

```
<!-- Activity that allows the user to send new SMS/MMS messages -->
<activity android:name=".ComposeSmsActivity" >
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <action android:name="android.intent.action.SENDTO" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
        <data android:scheme="mms" />
        <data android:scheme="mmsto" />
    </intent-filter>
</activity>
```

That may not be terribly surprising. What *is* surprising is that Google also expects you to have an exported *service* for handling "quick response" requests. These requests come when the user receives a phone call and taps on an icon to reply with a text message, rather than accept the call. In those cases, Android will invoke a service in the default SMS client, with an action of `android.intent.action.RESPOND_VIA_MESSAGE`. The `Intent` that you receive in `onStartCommand()` (or `onHandleIntent()`, if you elect to use an `IntentService`) will have an `EXTRA_TEXT` and optionally an `EXTRA_SUBJECT` as extras, representing the message to be sent. The `Uri` in the `Intent` will indicate the intended recipient of the message. Your job is to use `SmsManager` to actually send the message.

The Android documentation cites this as the relevant `<service>` element:

```
<!-- Service that delivers messages from the phone "quick response" -->
<service android:name=".HeadlessSmsSendService"
         android:permission="android.permission.SEND_RESPOND_VIA_MESSAGE"
         android:exported="true" >
    <intent-filter>
        <action android:name="android.intent.action.RESPOND_VIA_MESSAGE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
        <data android:scheme="mms" />
        <data android:scheme="mmsto" />
    </intent-filter>
</service>
```

Note:

- The `<service>` requires that the sender have the `SEND_RESPOND_VIA_MESSAGE` permission, to reduce spoofing

- The `android:exported="true"` shown in the sample should be superfluous, as since the `<service>` has an `<intent-filter>`, it should be exported by default
- The `<category>`, and possibly the `<data>`, elements may be erroneous... and since the author cannot find anything in the OS that uses `RESPOND_VIA_MESSAGE`, the author cannot validate that these elements should be here or represent copy-and-paste errors in the documentation

Handling Both Receive Options

If you want to support receiving SMS using both the legacy approach and the Android 4.4+ approach, you can have two `BroadcastReceiver` implementations, one for `android.provider.Telephony.SMS_RECEIVED` and one for `android.provider.Telephony.SMS_DELIVER`. However, you will only need the latter one on Android 4.4, and by default you would receive both broadcasts.

To handle that, you can define a boolean resource in the `res/values-v19/` directory (e.g., `isPreKitKat`) to be `false`, with a default definition in `res/values/` of `true` for the same resource. Then, in your manifest, you can have `android:enabled="@bool/isPreKitKat"` on your `SMS_RECEIVED` `<receiver>` element. This will only enable this component on API Level 18 and below, disabling it on API Level 19+.

You can also define a counterpart resource for the positive case (e.g., `@bool/isKitKat`), and use that to selectively enable the SMS and MMS receivers, if desired.

The SMS Inbox

Many users keep their text messages around, at least for a while. These are stored in an "inbox", represented by a `ContentProvider`. How you work with this `ContentProvider` — or if you can work with it at all, varies upon whether you are running on Android 4.4+ or not.

The Undocumented, Unsupported, Pre-Android 4.4 Way

When perusing the Internet, you will find various blog posts and such referring to the SMS inbox `ContentProvider`, represented by the `content://sms/inbox` `Uri`.

This `ContentProvider` is undocumented and is not part of the Android SDK, because it is not part of the Android OS.

Rather, this ContentProvider is used by the aforementioned Messaging application, for storing saved SMS messages. And, as noted, this application may or may not exist on any given Android device. If a device manufacturer replaces Messaging with their own application, there may be nothing on that device that responds to that Uri, or the schemas may be totally different. Plus, Android may well change or even remove this ContentProvider in future editions of Android.

For all those reasons, developers should not be relying upon this ContentProvider.

The Android 4.4+ Way

Android 4.4 has exposed a series of ContentProviders, in the android.provider.Telephony namespace, for storing SMS and MMS messages. These include:

- the Inbox for received messages
- the Outbox for a log of sent messages
- the Draft for messages that were written but have not yet been sent
- etc.

Some are duplicated, such as separate providers for the SMS inbox versus the MMS inbox. Some are distinct, such as Sms.Conversations and Mms.Rate.

All are largely undocumented.

The user's chosen default SMS client can write to these providers. Apps with READ_SMS permission should be able to read from them.

Asking to Change the Default

There are many areas in Android where the user must do two things to use an app:

1. Install the app (from the Play Store or elsewhere)
2. Go into Settings (or sometimes elsewhere) and indicate that a certain capability of the newly-installed app should become active

You see this with app widgets, input method editors, device administrators, and many others.

On Android 4.4+, you also see this with SMS/MMS clients. Devices usually ship with one. If the user wants a replacement, the user must indicate in Settings that this new SMS/MMS client should be the default, so it can write to the SMS/MMS `ContentProvider` suite.

Your app can determine what the default client is by calling `getDefaultSmsPackage()` on the `Telephony.Sms` class. This will return the package name of the current default client.

If this is not your package, and you would like the user to make you the default, you can start an activity to request this change:

```
Intent i = new Intent(Sms.Intents.ACTION_CHANGE_DEFAULT);
i.putExtra(Sms.Intents.EXTRA_PACKAGE_NAME, getPackageName());
startActivity(i);
```

The `EXTRA_PACKAGE_NAME` will trigger the UI to ask the user if the user wishes to change the current default to your package (versus anything else on the device that might also be a possible SMS/MMS client).

Hence, the recommended flow for a backup/restore app is to:

- Make note of the current default, via `getDefaultSmsPackage()`
- Request to the user to make you the default, via `ACTION_CHANGE_DEFAULT`
- Confirm that they did this, via `getDefaultSmsPackage()`
- If they did, do your backup or restore work
- Request to the user to restore the original default, via `ACTION_CHANGE_DEFAULT`

SMS and the Emulator

The “Emulator Control” view in DDMS allows you to send fake SMS messages to a running emulator. This is very useful for light testing.

You can also send fake SMS messages to an emulator via the emulator console. This can be accessed via **`telnet`**, where the console is available on `localhost` on your development machine, via the port number that appears in the title bar of your emulator window (e.g., `5554`). In the **`telnet`** session, you can enter **`sms send [sendingNumber> <txt>`**, replacing `<sendingNumber>` with the phone number of the pretend sender of the SMS, and replacing `<txt>` with the text message itself.

SMS Tokens

The changes that Android made in 4.4 to limit who can delete SMS messages all but eliminated one common bit of app functionality: sending an SMS from the server to a user's device to validate that the user's phone number is what was expected. This is still possible, but the app can no longer delete that SMS message, meaning that it will clutter up the user's SMS inbox.

Android 8.0+ provides "app-specific SMS tokens", via a `createAppSpecificSmsToken()` method on `SmsManager`. You supply a `PendingIntent`, and you get a unique string back. If an SMS is received by the device containing that string, the `PendingIntent` is invoked, instead of the message being delivered to the user's SMS client.

Hence, phone number validation flow is once again possible:

- Call `createAppSpecificSmsToken()` on the `SmsManager`
- Send that token, along with the phone number, to your server
- Have your server trigger an SMS message containing that token, sent to the device

Your `PendingIntent` that you gave to `createAppSpecificSmsToken()` will get triggered as a result, invoking whatever component that you identified in the underlying `Intent`.

The `SMS/Token` sample project demonstrates this flow. It consists of two activities: one to show you a generated token, and one that will be displayed when that token is received in an SMS message.

`MainActivity` is responsible for showing you the token:

```
package com.commonsware.android.sms.token;

import android.app.Activity;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.widget.TextView;

public class MainActivity extends Activity {
  @Override
```

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    SmsManager mgr=SmsManager.getDefault();
    String token=mgr.createAppSpecificSmsToken(buildPendingIntent());
    TextView tv=(TextView)findViewById(R.id.text);

    tv.setText(getString(R.string.msg, token));
  }

  private PendingIntent buildPendingIntent() {
    return(PendingIntent.getActivity(this, 1337,
      new Intent(this, ResultActivity.class), 0));
  }
}
```

(from SMS/Token/app/src/main/java/com/commonsware/android/sms/token/MainActivity.java)

Here, we:

- Get the `SmsManager` by calling the `getDefault()` `static` method
- Build an activity `PendingIntent` identifying `ResultActivity`
- Generate an SMS token associated with that `PendingIntent`
- Display that token as part of a message on the screen

The token itself is not really designed for manual user entry:

Figure 871: Token Sample App, Showing a Token

If another device sends an SMS message containing that token (along with perhaps other information), `ResultActivity` will be displayed:

```
package com.commonsware.android.sms.token;

import android.app.Activity;
import android.app.PendingIntent;
import android.os.Bundle;
import android.provider.Telephony;
import android.telephony.SmsManager;
import android.telephony.SmsMessage;
import android.widget.TextView;

public class ResultActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    TextView tv=(TextView)findViewById(R.id.text);

    for (SmsMessage pdu :
      Telephony.Sms.Intents.getMessagesFromIntent(getIntent())) {
```

```
      tv.append(pdu.getDisplayMessageBody());
    }
  }
}
```

(from SMS/Token/app/src/main/java/com/commonsware/android/sms/token/ResultActivity.java)

The actual SMS message is included in the `Intent` extras, filled into a copy of the `Intent` that you supplied in the `PendingIntent`. The `getMessagesFromIntent()` method on `Telephony.Sms.Intents` offers a convenient way to get the actual `SmsMessage` objects. Here, we assume that they represent a text message, and we concatenate their messages together to display in a `TextView`:

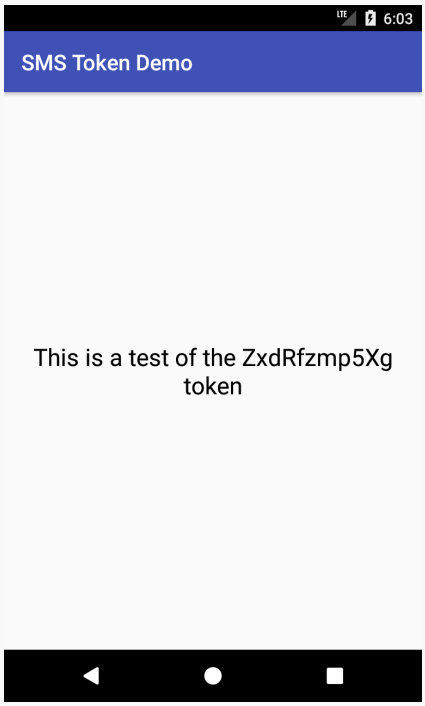


Figure 872: Token Sample App, Showing a Received Message with the Token

Usually, rather than show the token to the user, you will send it programmatically where it needs to go (e.g., a Web service call). Since the SMS message containing the token does not wind up in the user's SMS client, the message containing the token does not need to be human-readable. It does need to contain the token verbatim, without any compression, encryption, or other conversions placed upon it.

Also:

- There can only be one outstanding token per app per device. If you call `createAppSpecificSmsToken()` twice in succession, the first token will be invalidated and ignored.
- A token is only good for one message. If you try sending messages to the device twice with the same token, the second message will be delivered to the user's SMS client, rather than invoke your `PendingIntent`.
- For a demo like this sample app, hardware is not required. You can use the emulator's extended controls to send a fake SMS message to the emulator, and that will go through normal SMS processing, including token analysis.

NFC

NFC, courtesy of high-profile boosters like Google Wallet, is poised to be a significant new capability in Android devices. While at the time of this writing, only a handful of Android devices have NFC built in, other handsets are slated to be NFC-capable in the coming months. Google is hoping that developers will write NFC-aware applications to help further drive adoption of this technology by device manufacturers.

This, of course, raises the question: what is NFC? Besides being where the Green Bay Packers play, that is?

(For those of you from outside of the United States, that was an American football joke. We now return you to your regularly-scheduled chapter.)

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on broadcast `Intents` and services.

What Is NFC?

NFC stands for Near-Field Communications. It is a wireless standard for data exchange, aimed at very short range transmissions — on the order of a couple of centimeters. NFC is in wide use today, for everything from credit cards to passports. Typically, the NFC data exchange is for simple data — contact information, URLs, and the like.

In particular, NFC tends to be widely used where one side of the communications channel is "passive", or unpowered. The other side (the "initiator") broadcasts a signal, which the passive side converts into power enough to send back its response. As such, NFC "tags" containing such passive targets can be made fairly small and can be embedded in a wide range of containers, from stickers to cards to hats.

The objective is "low friction" interaction — no pairing like with Bluetooth, no IP address shenanigans as with WiFi. The user just taps and goes.

… Compared to RFID?

NFC is often confused with or compared to RFID. It is simplest to think of RFID as being an umbrella term, under which NFC falls. Not every RFID technology is NFC, but many things that you hear of being "RFID" may actually be NFC-compliant devices or tags.

… Compared to QR Codes?

In many places, NFC will be used in ways you might consider using QR codes. For example, a restaurant could use either technology, or both, on a sign to lead patrons to the restaurant's Yelp page, as a way of soliciting reviews. Somebody with a capable device could either tap the NFC tag on the sign to bring up Yelp or take a picture of the QR code and use that to bring up Yelp.

NFC's primary advantage over QR codes is that it requires no user intervention beyond physically moving their device in close proximity to the tag. QR codes, on the other hand, require the user to launch a barcode scanning application, center the barcode in the viewfinder, and then get the results. The net effect is that NFC will be faster.

QR's advantages include:

1. No need for any special hardware to generate the code, as opposed to needing a tag and something to write information into the tag for NFC
2. The ability to display QR codes in distant locations (e.g., via Web sites), whereas NFC requires physical proximity

To NDEF, Or Not to NDEF

RFID is a concept, not a standard. As such, different vendors created their own ways of structuring data on these tags or chips, making one vendor's tags incompatible with another vendor's readers or writers. While various standards bodies, like ISO, have gotten involved, it's still a bit of a rat's nest of conflicting formats and approaches.

The NFC offshoot of RFID has had somewhat greater success in establishing standards. NFC itself is an ISO and ECMA standard, covering things like transport protocols and transfer speeds. And a consortium called the NFC Forum created NDEF — the NFC Data Exchange Format — for specifying the content of tags.

However, not all NFC tags necessarily support NDEF. NDEF is much newer than NFC, and so lots of NFC tags are out in the wild that were distributed before NDEF even existed.

You can roughly divide NFC tags into three buckets:

- Those that support NDEF "out of the box"
- Those that can be "formatted" as NDEF
- Those that use other content schemes

Android has some support for non-NDEF tags, such as the MIFARE Classic. However, the hope and expectation going forward is that NFC tags will coalesce around NDEF.

NDEF, as it turns out, maps neatly to Android's `Intent` system, as you will see as we proceed through this chapter.

NDEF Modalities

Most developers interested in NFC will be interested in reading NFC tags and retrieving the NDEF data off of them. In Android, tapping an NDEF tag with an NFC-capable device will trigger an activity to be started, based on a certain `IntentFilter`.

Some developers will be interested in writing to NFC tags, putting URLs, vCards, or other information on them. This may or may not be possible for any given tag.

And while the "traditional" thinking around NFC has been that one side of the communication is a passive tag, Android will help promote the "peer-to-peer" approach — having two Android devices exchange data via NFC and NDEF. Basically, putting the two devices back-to-back will cause each to detect the other device's "tag", and each can read and write to the other via this means. This is referred to as "Android Beam" and will be discussed later in this chapter.

Of course, all of these are only available on hardware. At the present time, there is no emulator for NFC, nor any means of accessing a USB NFC reader or writer from the emulator.

NDEF Structure and Android's Translation

NDEF is made up of messages, themselves made up of a series of records. From Android's standpoint, each tag consists of one such message.

Each record consists of a binary (`byte` array) payload plus metadata to describe the nature of the payload. The metadata primarily consists of a type and a subtype. There are quite a few combinations of these, but the big three for new Android NFC uses are:

- A type of `TNF_WELL_KNOWN` and a subtype of `RTD_TEXT`, indicating that the payload is simply plain text
- A type of `TNF_WELL_KNOWN` and a subtype of `RTD_URI`, indicating that the payload is a URI, such as a URL to a Web page
- A type of `TNF_MIME_MEDIA`, where the subtype is a standard MIME type, indicating that the payload is of that MIME type

When Android scans an NDEF tag, it will use this information to construct a suitable `Intent` to use with `startActivity()`. The action will be `android.nfc.action.NDEF_DISCOVERED`, to distinguish the scanned-tag case from, say, something simply asking to view some content. The MIME type in the Intent will be `text/plain` for the first scenario above or the supplied MIME type for the third scenario above. The data (`Uri`) in the Intent will be the supplied URI for the second scenario above. Once constructed, Android will invoke `startActivity()` on that `Intent`, bringing up an activity or an activity chooser, as appropriate.

NFC-capable Android devices have a Tags application pre-installed that will handle any NFC tag not handled by some other app. So, for example, an NDEF tag with an

HTTP URL will fire up the Tags application, which in turn will allow the user to open up a Web browser on that URL.

The Reality of NDEF

The enthusiasm that some have with regards to Android and NFC technology needs to be tempered by the reality of NDEF, NFC tags in general, and Android's support for NFC. It is easy to imagine all sorts of possibilities that may or may not be practical when current limitations are reached.

Some Tags are Read-Only

Some tags come "from the factory" read-only. Either you arrange for the distributor to write data onto them (e.g., blast a certain URL onto a bunch of NFC stickers to paste onto signs), or they come with some other pre-established data. Touchatag, for example, distributes NFC tags that have Touchatag URLs on them — they then help you set up redirects from their supplied URL to ones you supply.

While these tags will be of interest to consumers and businesses, they are unlikely to be of interest to Android developers, since their use cases are already established and typically do not need custom Android application support. Android developers seeking customizable tags will want ones that are read-write, or at least write-once.

Some Tags Can't Be Read-Only

Conversely, some tags lack any sort of read-only flag. An ideal tag for developers is one that is write-once: putting an NDEF message on the tag and flagging it read-only in one operation. Some tags do not support this, or making the tag read-only at any later point. The MIFARE Classic 1K tag is an example — while technically it can be made read-only, it requires a key known only to the tag manufacturer.

Some Tags Need to be Formatted

The MIFARE Classic 1K NFC tag is NDEF-capable, but must be "formatted" first, supplying the initial NDEF message contents. You have the option of formatting it read-write or read-only (turning the Classic 1K a write-once tag).

This is not a problem — in fact, the write-once option may be compelling. However, it is something to keep in mind.

Also, note that the MIFARE Classic 1K, while it can be formatted as NDEF, uses a proprietary protocol "under the covers". Not all Android devices will support the Classic 1K, as the device manufacturers elect not to pay the licensing fee. Where possible, try to stick to tags that are natively NDEF-compliant (so-called "NFC Forum Tag Types 1-4").

Tags Have Limited Storage

The "1K" in the name "MIFARE Classic 1K" refers to the amount of storage on the tag: 1 kilobyte of information.

And that's far larger than other tags, such as the MIFARE Ultralight C, some of which have ~64 bytes of storage.

Clearly, you will not be writing an MP3 file or JPEG photo to these tags. Rather, the tags will tend to either be a "launcher" into something with richer communications (e.g., URL to a Web site) or will use the sorts of data you may be used to from QR codes, such as a vCard or iCalendar for contact and event data, respectively.

NDEF Data Structures Are Documented Elsewhere

The Android developer documentation is focused on the Android classes related to NFC and on the `Intent` mechanism used for scanned tags. It does not focus on the actual structure of the payloads.

For `TNF_MIME_MEDIA` and `RTD_TEXT`, the payload is whatever you want. For `RTD_URI`, however, the byte array has a bit more structure to it, as the NDEF specification calls for a single byte to represent the URI prefix (e.g., `http://www.` versus `http://` versus `https://www.`). The objective, presumably, is to support incrementally longer URLs on tags with minuscule storage. Hence, you will need to convert your URLs into this sort of byte array if you are writing them out to a tag.

Generally speaking, the rules surrounding the structure of NDEF messages and records is found at the NFC Forum site.

Tag and Device Compatibility

Different devices will have different NFC chipsets. Not all NFC chipsets can read and write all tags. The expectation is that NDEF-formatted tags will work on all devices,

but if you wander away from that, things get dicier. For example, NXP's Mifare Classic tag can only be read and written by NXP's NFC chip.

This is increasingly a challenge for Android developers, as a Broadcom NFC chip is becoming significantly more popular. Many new major Android devices, such as the Samsung Galaxy S4, the Nexus 4, the Nexus 10, and the 2013/2nd generation version of the Nexus 7, all use the Broadcom chip. Those devices are incompatible with the Mifare tags, such as the popular Mifare Classic 1K.

That is because NXP is the maker of the Mifare Classic series, and those tags broke the NFC Forum's standards to create a tag that was NXP-specific.

Right now, NTAG203 and Topaz tags (like the Topaz 512), are likely candidate tags that will work across all NFC-capable Android devices, due to their adherence to NFC standard protocols.

Sources of Tags

NFC tags are not the sort of thing you will find on your grocer's shelves. In fact, few, if any, mainstream firms sell them today.

Here are some online sites from which you can order rewritable NFC tags, listed here in alphabetical order:

1. Andytags
2. Buy NFC Tags
3. Smartcard Focus
4. tagstand

Note that not all may ship to your locale.

Writing to a Tag

So, let's see what it takes to write an NDEF message to a tag, formatting it if needed. The code samples shown in this chapter are from the `NFC/URLTagger` sample application. This application will set up an activity to respond to `ACTION_SEND` activity `Intents`, with an eye towards receiving a URL from a browser, then waiting for a tag and writing the URL to that tag. The idea is that this sort of application could be used by non-technical people to populate tags containing URLs to their company's Web site, etc.

Getting a URL

First, we need to get a URL from the browser. As we saw in the chapter on integration, the standard Android browser uses `ACTION_SEND` of `text/plain` contents when the user chooses the "Share Page" menu. So, we have one activity, `URLTagger`, that will respond to such an `Intent`:

```
    <activity
      android:name="URLTagger"
      android:label="@string/app_name">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

(from NFC/URLTagger/app/src/main/AndroidManifest.xml)

Of course, lots of other applications support `ACTION_SEND` of `text/plain` contents that are not URLs. A production-grade version of this application would want to validate the `EXTRA_TEXT` `Intent` extra to confirm that, indeed, this is a URL, before putting in an NDEF message claiming that it is a URL.

Detecting a Tag

When the user shares a URL with our application, our activity is launched. At that point, we need to go into "detect a tag" mode – the user should then tap their device to a tag, so we can write out the URL.

First, in `onCreate()`, we get access to the `NfcAdapter`, which is our gateway to much of the NFC functionality in Android:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  nfc=NfcAdapter.getDefaultAdapter(this);
}
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

We use a boolean data member — `inWriteMode` — to keep track of whether or not we are set up to write to a tag. Initially, of course, that is set to be `false`. Hence, when we are first launched, by the time we get to `onResume()`, we can go ahead and register our interest in future tags:

```
@Override
public void onResume() {
  super.onResume();

  if (!inWriteMode) {
    IntentFilter discovery=new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    IntentFilter[] tagFilters=new IntentFilter[] { discovery };
    Intent i=new Intent(this, getClass())
              .addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP|
                        Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

    inWriteMode=true;
    nfc.enableForegroundDispatch(this, pi, tagFilters, null);
  }
}
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

When an NDEF-capable tag is within signal range of the device, Android will invoke `startActivity()` for the `NfcAdapter.ACTION_TAG_DISCOVERED` `Intent` action. However, it can do this in one of two ways:

- Normally, it will use a chooser (via `Intent.createChooser()`) to allow the user to pick from any activities that claim to support this action.
- The foreground application can request via `enableForegroundDispatch()` for it to handle all tag events while it is in the foreground, superseding the normal `startActivity()` flow. In this case, while Android still will invoke an activity, it will be our activity, not any other one.

We want the second approach right now, so the next tag brought in range is the one we will try writing to.

To do that, we need to create an array of `IntentFilter` objects, identifying the NFC-related actions that we want to capture in the foreground. In this case, we only care about `ACTION_TAG_DISCOVERED` – if we were supporting non-NDEF NFC tags, we might also need to watch for `ACTION_TECH_DISCOVERED`.

We also need a `PendingIntent` identifying the activity that should be invoked when such a tag is encountered while we are in the foreground. Typically, this will be the current activity. By adding `FLAG_ACTIVITY_SINGLE_TOP` and

`FLAG_ACTIVITY_CLEAR_TOP` to the `Intent` as flags, we ensure that our current specific instance of the activity will be given control again via `onNewIntent()`.

Armed with those two values, we can call `enableForegroundDispatch()` on the `NfcAdapter` to register our request to process tags via the current activity instance.

In `onPause()`, if the activity is finishing, we call `disableForegroundDispatch()` to undo the work done in `onResume()`:

```
@Override
public void onPause() {
  if (isFinishing()) {
    nfc.disableForegroundDispatch(this);
    inWriteMode=false;
  }

  super.onPause();
}
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

We have to see if we are finishing, because even though our activity never leaves the screen, Android still calls `onPause()` and `onResume()` as part of delivering the `Intent` to `onNewIntent()`. Our approach, though, has flaws — if the user presses HOME, for example, we never disable the NFC dispatch logic. A production-grade application would need to handle this better.

For any of this code to work, we need to hold the NFC permission via an appropriate line in the manifest:

```
<uses-permission android:name="android.permission.NFC"/>
```

Also note that if you have several activities that the user can reach while you are trying to also capture NFC tag events, you will need to call `enableForegroundDispatch()` in each activity — it's a per-activity request, not a per-application request.

Reacting to a Tag

Once the user brings a tag in range, `onNewIntent()` will be invoked with the `ACTION_TAG_DISCOVERED` `Intent` action:

```
@Override
protected void onNewIntent(Intent intent) {
  if (inWriteMode &&
      NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
    Tag tag=intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    byte[] url=buildUrlBytes(getIntent().getStringExtra(Intent.EXTRA_TEXT));
    NdefRecord record=new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
                                     NdefRecord.RTD_URI,
                                     new byte[] {}, url);
    NdefMessage msg=new NdefMessage(new NdefRecord[] {record});

    new WriteTask(this, msg, tag).execute();
  }
}
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

If we are in write mode and the delivered `Intent` is indeed an `ACTION_TAG_DISCOVERED` one, we can get at the `Tag` object associated with the user's NFC tag via the `NfcAdapter.EXTRA_TAG` `Parcelable` extra on the `Intent`.

Writing an NDEF message to the tag, therefore, is a matter of crafting the message and actually writing it. An NDEF message consists of one or more records (though, typically, only one record is used), with each record wrapping around a byte array of payload data.

Getting the Shared URL

We did not do anything to get the URL out of the `Intent` back in `onCreate()`, when our activity was first started up. Now, of course, we need that URL. You might think it is too late to get it, since our activity was effectively started again due to the tag and `onNewIntent()`.

However, `getIntent()` on an `Activity` always returns the `Intent` used to create the activity in the first place. The `getIntent()` value is not replaced when `onNewIntent()` is called.

Hence, as part of the `buildUrlBytes()` method to create the binary payload, we can go and call `getIntent().getStringExtra(Intent.EXTRA_TEXT)` to retrieve the URL.

Creating the Byte Array

Given the URL, we need to convert it into a byte array suitable for use in a `TNF_WELL_KNOWN`, `RTD_URI` NDEF record. Ordinarily, you would just call `toByteArray()` on the `String` and be done with it. However, the byte array we need uses a single byte to indicate the URL prefix, with the rest of the byte array for the characters after this prefix.

This is efficient. This is understandable. This is annoying.

First, we need the roster of prefixes, defined in `URLTagger` as a static data member cunningly named `PREFIXES`:

```
static private final String[] PREFIXES={"http://www.", "https://www.",
                                        "http://", "https://",
                                        "tel:", "mailto:",
                                        "ftp://anonymous:anonymous@",
                                        "ftp://ftp.", "ftps://",
                                        "sftp://", "smb://",
                                        "nfs://", "ftp://",
                                        "dav://", "news:",
                                        "telnet://", "imap:",
                                        "rtsp://", "urn:",
                                        "pop:", "sip:", "sips:",
                                        "tftp:", "btspp://",
                                        "btl2cap://", "btgoep://",
                                        "tcpobex://",
                                        "irdaobex://",
                                        "file://", "urn:epc:id:",
                                        "urn:epc:tag:",
                                        "urn:epc:pat:",
                                        "urn:epc:raw:",
                                        "urn:epc:", "urn:nfc:"};
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

Then, in `buildUrlBytes()`, we need to find the prefix (if any) and use it:

```
private byte[] buildUrlBytes(String url) {
  byte prefixByte=0;
  String subset=url;
  int bestPrefixLength=0;

  for (int i=0;i<PREFIXES.length;i++) {
    String prefix = PREFIXES[i];
```

```
      if (url.startsWith(prefix) && prefix.length() > bestPrefixLength) {
        prefixByte=(byte)(i+1);
        bestPrefixLength=prefix.length();
        subset=url.substring(bestPrefixLength);
      }
    }

    final byte[] subsetBytes = subset.getBytes();
    final byte[] result = new byte[subsetBytes.length+1];

    result[0]=prefixByte;
    System.arraycopy(subsetBytes, 0, result, 1, subsetBytes.length);

    return(result);
  }
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

We iterate over the `PREFIXES` array and find a match, if any, and the best possible match if there is more than one. If there is a match, we record the NDEF value for the first byte (our `PREFIXES` index plus one) and create a subset string containing the characters after the prefix. If there is no matching prefix, the prefix byte is `0` and we will include the full URL.

Given that, we construct a byte array containing our prefix byte in the first slot, and the rest taken up by the byte array of the subset of our URL.

Creating the NDEF Record and Message

Given the result of `buildUrlBytes()`, our `onNewIntent()` implementation creates a `TNF_WELL_KNOWN`, `RTD_URI` `NdefRecord` object, and pours that into an `NdefMessage` object.

The third parameter to the `NdefRecord` constructor is a byte array representing the optional "ID" of this record, which is not necessary here.

Finally, we delegate the actual writing to a `WriteTask` subclass of `AsyncTask`, as writing the `NdefMessage` to the `Tag` is... interesting.

Writing to a Tag

Here is the aforementioned `WriteTask` static inner class:

```
static class WriteTask extends AsyncTask<Void, Void, Void> {
  Activity host=null;
  NdefMessage msg=null;
  Tag tag=null;
  String text=null;

  WriteTask(Activity host, NdefMessage msg, Tag tag) {
    this.host=host;
    this.msg=msg;
    this.tag=tag;
  }

  @Override
  protected Void doInBackground(Void... arg0) {
    int size=msg.toByteArray().length;

    try {
      Ndef ndef=Ndef.get(tag);

      if (ndef==null) {
        NdefFormatable formatable=NdefFormatable.get(tag);

        if (formatable!=null) {
          try {
            formatable.connect();

            try {
              formatable.format(msg);
            }
            catch (Exception e) {
              text="Tag refused to format";
            }
          }
          catch (Exception e) {
            text="Tag refused to connect";
          }
          finally {
            formatable.close();
          }
        }
        else {
          text="Tag does not support NDEF";
        }
      }
      else {
        ndef.connect();

        try {
```

```
          if (!ndef.isWritable()) {
            text="Tag is read-only";
          }
          else if (ndef.getMaxSize()<size) {
            text="Message is too big for tag";
          }
          else {
            ndef.writeNdefMessage(msg);
          }
        }
        catch (Exception e) {
          text="Tag refused to connect";
        }
        finally {
          ndef.close();
        }
      }
    }
    catch (Exception e) {
      Log.e("URLTagger", "Exception when writing tag", e);
      text="General exception: "+e.getMessage();
    }

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    if (text!=null) {
      Toast.makeText(host, text, Toast.LENGTH_SHORT).show();
    }

    host.finish();
  }
}
```

(from NFC/URLTagger/app/src/main/java/com/commonsware/android/nfc/url/URLTagger.java)

In `doInBackground()`, after making note of how big the message is in bytes, we first try to get the `Ndef` aspect of the `Tag` object, by calling the static `get()` method on the `Ndef` class. If the tag is an NDEF tag, this should return an `Ndef` instance. If it does not, we try to get an `NdefFormatable` aspect by calling `get()` on the `NdefFormatable` class. If the tag is not NDEF now but can be formatted as NDEF, this should give us an `NdefFormatable` object. If both aspect attempts fail, we bail out, displaying a `Toast` to let the user know that while the tag they used is NFC, it is not NDEF-compliant.

If the tag turned out to be `NdefFormatable`, to put the `NdefMessage` on it, we first `connect()` to the tag, then `format()` it, supplying the message. `NdefFormatable` also supports `formatReadOnly()` for tags that support that mode — this will write the message on the tag, then block it from further updates. When we are done, we `close()` the connection.

If the tag turned out to be `Ndef` already, we `connect()` to it, then see if it is writable and has enough room. If it meets both of those criteria, we can emit the message via `writeNdefMessage()`, which overwrites the NDEF message that had already existed on the tag (if any). If the tag supported it, a call to `makeReadOnly()` would block further updates to the tag. Again, when we are done, we `close()` the connection.

All of the actual NFC I/O is performed in `doInBackground()`, because this I/O may take some time, and we do not want to block the main application thread while doing it.

Responding to a Tag

Writing to a tag is a bit complicated. Responding to an NDEF message on a tag is significantly easier.

If the foreground activity is not consuming NFC events — as `URLTagger` does in write mode — then Android will use normal `Intent` resolution with `startActivity()` to handle the tag. To respond to the tag, all you need to do is have an activity set up to watch for an `android.nfc.action.NDEF_DISCOVERED` `Intent`. To get control ahead of the built-in Tags application, also have a `<data>` element that describes the sort of content or URL you are expecting to find on the tag.

For example, suppose you used the Android browser to visit some page on the CommonsWare Web site, and you wrote that to a tag using `URLTagger`. The URLTagger application has another activity, `URLHandler`, that will respond when you tap the newly-written tag from the home screen or anywhere else. It accomplishes this via a suitable `<intent-filter>`:

```
    <activity
      android:name="URLHandler"
      android:label="@string/app_name">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <data
```

```
        android:host="commonsware.com"
        android:scheme="http"/>

      <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
  </activity>
```

(from NFC/URLTagger/app/src/main/AndroidManifest.xml)

The `URLHandler` activity can then use `getIntent()` to retrieve the key pieces of data from the tag itself, if needed. In particular, the `EXTRA_NDEF_MESSAGES` `Parcelable` array extra will return an array of `NdefMessage` objects. Typically, there will only be one of these. You can call `getRecords()` on the `NdefMessage` to get at the array of `NdefRecord` objects (again, typically only one). Methods like `getPayload()` will allow you to get at the individual portions of the record.

The nice thing is that the URL still works, even if URLTagger is not on the device. In that case, the Tags application would react to the tag, and the user could tap on it to bring up a browser on this URL. A production application might create a Web page that tells the user about this great and wonderful app she can install, and provide links to the Play Store (or elsewhere) to go get the app.

Expected Pattern: Bootstrap

Tags tend to have limited capacity. Even in peer-to-peer settings, the effective bandwidth of NFC is paltry compared to anything outside of dial-up Internet access.

As a result, NFC will be used infrequently as the complete communications solution between a publisher and a device. Sometimes it will, when the content is specifically small, such as a contact (vCard) or event (iCalendar). But, for anything bigger than that, NFC will serve more as a convenient bootstrap for more conventional communications options:

1. Embedding a URL in a tag, as the previous sample showed, allows an installed application to run or a Web site to be browsed
2. Embedding a Play Store URL in a tag allows for easy access to some specialized app (e.g., menu for a restaurant)
3. A multi-player game might use peer-to-peer NFC to allow local participants to rapidly connect into the same shared game area, where the game is played over the Internet or Bluetooth
4. And so on.

Mobile Devices are Mobile

Reading and writing NFC tags is a relatively slow process, mostly due to low bandwidth. It may take a second or two to actually complete the operation.

Users, however, are not known for their patience.

If a user moves their device out of range of the tag while Android is attempting to read it, Android simply will skip the dispatch. If, however, the tag leaves the signal area of the device while you are writing to it, you will get an `IOException`. At this point, the state of the tag is unknown.

You may wish to incorporate something into your UI to let the user know that you are working with the tag, encouraging them to leave the phone in place until you are done.

Enabled and Disabled

There are two separate system settings that control NFC behavior:

- The user could have NFC disabled outright, which you would detect by calling `isEnabled()` on your `NfcAdapter`
- The user could have NFC enabled but have Android Beam disabled, which you would detect by calling `isNdefPushEnabled()` on your `NfcAdapter`

As with most enabled/disabled settings, you cannot change these values yourself. On newer Android SDK versions, though, you can try to bring up the relevant Settings screens for the user to enable these features, by using the following activity action strings from the `android.provider.Settings` class:

- `ACTION_NFC_SETTINGS` for the main NFC settings screen (added in API Level 16)
- `ACTION_NFCSHARING_SETTINGS` for the Android Beam settings screen (added in API Level 14)

Android Beam

Android Beam is Google's moniker for peer-to-peer NFC messaging, with an emphasis — obviously — on Android apps. Rather than you tapping your NFC-

capable Android device on a smart tag, you put it back-to-back with another NFC-capable Android device, and romance ensues.

Partially, this is simply one side of the exchange "pushing" an NDEF record, in a fashion that makes the other side of the exchange think that it is picking up a smart tag.

Partially, this is the concept of the "Android Application Record" (AAR), another NDEF record you can place in the NDEF message being pushed. This will identify the app you are trying to push the message to. If nothing on the device can handle the rest of the NDEF message, the AAR will lead Android to start up an app, or even lead the user to the Play Store to go download said app.

As the basis for explaining further how this all works, let's take a look at the NFC/WebBeam sample application. The UI consists of a `WebViewFragment`, in which we can browse to some Web page. Then, running this app on two NFC-capable devices, one app can "push" the URL of the currently-viewed Web page to the other app, which will respond by displaying that page. In this fashion, we are "sharing" a URL, without one side having to type it in by hand. And, while we are using this to share a URL, you could use Android Beam to share any sort of bootstrapping data, such as the user IDs of each person, for use in connecting to some common game server.

The Fragment

The fragment that implements our UI, `BeamFragment`, extends from `WebViewFragment`. In `onActivityCreated()`, we configure the `WebView`, load up Google's home page, and indicate that would like to participate in the action bar (via a call to `setHasOptionsMenu()`):

```
@SuppressLint("SetJavaScriptEnabled")
@Override
public void onActivityCreated(Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);

  getWebView().setWebViewClient(new BeamClient());
  getWebView().getSettings().setJavaScriptEnabled(true);
  loadUrl("http://google.com");
  setHasOptionsMenu(true);
}
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/BeamFragment.java)

To keep all links within the `WebView`, we attached a `WebViewClient` implementation, named `BeamClient`, that just loads all requested URLs back into the `WebView`:

```
class BeamClient extends WebViewClient {
  @Override
  public boolean shouldOverrideUrlLoading(WebView wv, String url) {
    wv.loadUrl(url);

    return(true);
  }
}
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/BeamFragment.java)

We add one item to the action bar: a toolbar button (`R.id.beam`) that will be used to indicate we wish to beam the URL in our `WebView` to another copy of this application running on another NFC-capable Android device:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  if (getContract().hasNFC()) {
    inflater.inflate(R.menu.actions, menu);
  }

  super.onCreateOptionsMenu(menu, inflater);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId() == R.id.beam) {
    getContract().enablePush();

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/BeamFragment.java)

So, when the app is initially launched, it will look something like this:

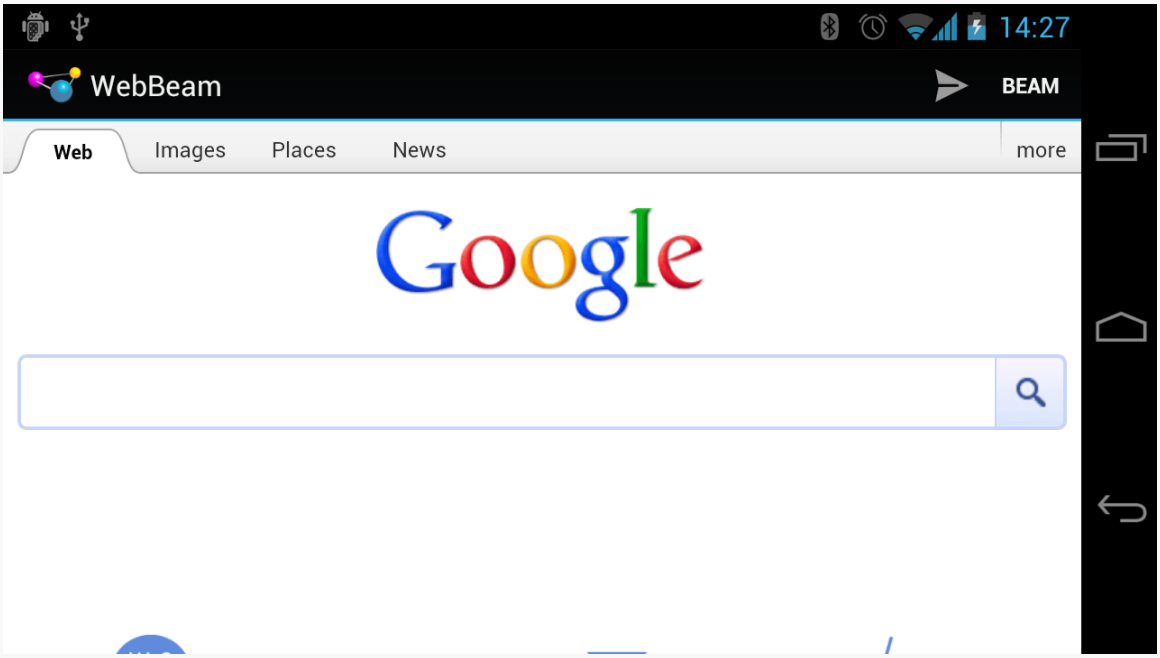


Figure 873: The WebBeam UI

The user can use Google to find a Web page worth beaming.

Requesting the Beam

Our hosting activity, `WebBeamActivity`, gets access to our `NfcAdapter`, as we did in the previous example:

```
    adapter=NfcAdapter.getDefaultAdapter(this);
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/WebBeamActivity.java)

When the user taps on our action bar item, the fragment calls `enablePush()` on the activity. `WebBeamActivity`, in turn, calls `setNdefPushMessageCallback()` on the `NfcAdapter`, supplying two parameters:

1. An implementation of the `NfcAdapter.CreateNdefMessageCallback` interface, used to let us know when another device is in range for us to beam to (in our case, `WebBeamActivity` implements this interface)
2. Our activity that is participating in this push

If something else comes to the foreground, `onStop()` will call a corresponding `disablePush()`, which also calls `setNdefPushMessageCallback()`, specifying a `null` first parameter, to turn off our request to beam:

```
void enablePush() {
  adapter.setNdefPushMessageCallback(this, this);
}

void disablePush() {
  adapter.setNdefPushMessageCallback(null, this);
}
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/WebBeamActivity.java)

In between the calls to `enablePush()` and `disablePush()`, if another NFC device comes in range that supports the NDEF push protocols, we're beamin'.

Sending the Beam

When our beam-enabled device encounters another beam-capable device, our `NfcAdapter.CreateNdefMessageCallback` is called with `createNdefMessage()`, where we need to prepare the `NfcMessage` to beam to the other party:

```
@Override
public NdefMessage createNdefMessage(NfcEvent arg0) {
  NdefRecord uriRecord=
      new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
                     MIME_TYPE.getBytes(Charset.forName("US-ASCII")),
                     new byte[0],
                     beamFragment.getUrl()
                         .getBytes(Charset.forName("US-ASCII")));
  NdefMessage msg=
      new NdefMessage(
                      new NdefRecord[] {
                          uriRecord,
                          NdefRecord.createApplicationRecord("com.commonsware.android.webbeam") });

  return(msg);
}
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/WebBeamActivity.java)

We first create a typical `NfcRecord`, in this case of `TNF_MIME_MEDIA`, with a MIME type defined in a static data member and payload consisting of the URL from our `WebView`:

```
private static final String MIME_TYPE=
    "application/vnd.commonsware.sample.webbeam";
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/WebBeamActivity.java)

You might wonder why we are using `TNF_MIME_MEDIA`, instead of `TNF_WELL_KNOWN` and a subtype of `RTD_URI`, since our payload is a URL. The reason is that we need to have a unique MIME type for our message for the whole beam process to work properly, and `TNF_WELL_KNOWN` does not support MIME types. This is also why the MIME type is something distinctive, and not just `text/plain` — it has to be something only we will pick up.

Our `NfcMessage` then consists of *two* `NfcRecord` objects: the one we just created, and one created via the static `createApplicationRecord()` method on `NfcRecord`. This helper method creates an AAR record, identifying our application by its Android package name. This record must go last – Android will try to find an app to work with based on the other records first, before "failing over" to use the AAR.

Receiving the Beam

To receive our beam, our `WebBeamActivity` must be configured in the manifest to respond to `NDEF_DISCOVERED` actions with our unique MIME type:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.webbeam"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.NFC"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name=".WebBeamActivity"
      android:label="@string/app_name"
      android:launchMode="singleTask"
      android:screenOrientation="landscape">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
```

```
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <category android:name="android.intent.category.DEFAULT"/>

        <data android:mimeType="application/vnd.commonsware.sample.webbeam"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from NFC/WebBeam/app/src/main/AndroidManifest.xml)

You will also notice that we set `android:launchMode="singleTask"` on this activity. That is so we will only have one instance of this activity, regardless of whether it is in the foreground or not. Otherwise, if we already have an instance of this activity, and we receive a beam, Android will create a *second* instance of this activity — when the user later presses BACK, they return to our first instance, and wonder why our app is broken.

If we receive the beam, we will get the `Intent` for the `NDEF_DISCOVERED` action *either* in `onCreate()` (if we were not already running) or `onNewIntent()` (if we were). In either case, we want to handle it the same way: pass the URL from the first record's payload to our `BeamFragment`. However, we cannot do that from `onCreate()` — the fragment will not have created the `WebView` yet. So, we use a trick: calling `post()` with a `Runnable` puts that `Runnable` on the *end* of the work queue for the main application thread. We can delay our processing of the `Intent` by this mechanism, so we can safely assume the `WebView` exists.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  beamFragment=
      (BeamFragment)getFragmentManager().findFragmentById(android.R.id.content);

  if (beamFragment == null) {
    beamFragment=new BeamFragment();

    getFragmentManager().beginTransaction()
                        .add(android.R.id.content, beamFragment)
                        .commit();
  }

  adapter=NfcAdapter.getDefaultAdapter(this);
```

```
    findViewById(android.R.id.content).post(new Runnable() {
      public void run() {
        handleIntent(getIntent());
      }
    });
  }

  @Override
  public void onNewIntent(Intent i) {
    handleIntent(i);
  }
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/WebBeamActivity.java)

```
  private void handleIntent(Intent i) {
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(i.getAction())) {
      Parcelable[] rawMsgs=
          i.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
      NdefMessage msg=(NdefMessage)rawMsgs[0];
      String url=new String(msg.getRecords()[0].getPayload());

      beamFragment.loadUrl(url);
    }
  }
```

(from NFC/WebBeam/app/src/main/java/com/commonsware/android/webbeam/WebBeamActivity.java)

The Scenarios

There are three possible scenarios, when we try beaming from one device to another:

1. The other device has our application installed, and it is running. In that case, our activity is brought to the foreground and the `Intent` is delivered to it, courtesy of our `NDEF_DISCOVERED` `<intent-filter>` with our unique MIME type.
2. The other device has our application installed, but it is not running. Android's `Intent` system handles this in the same general fashion as the first scenario, though it starts up a process for us and creates our activity instance anew in this case.
3. The other device does not have our application installed. Since nothing (hopefully) claims to support our unique MIME type, the AAR takes effect, and the user is led to the Play Store to go download our app (or, in this case, display an error message, as `WebBeam` is not in the Play Store).

Beaming Files

Android 4.1 (a.k.a., Jelly Bean) added in a far simpler facility for an app to beam a file to another device using the Android Beam system. You can use `setBeamPushUris()` or `setBeamPushUrisCallback()` on an `NfcAdapter` to hand Android one or more `Uri` objects representing files to be transferred. While the initial connection will be made via NFC and Android Beam, the actual data transfer will be via Bluetooth or WiFi, much more suitable than NFC for bulk data.

The difference between the two approaches is mostly when you provide the array of `Uri` objects. With `setBeamPushUris()`, you initiate the beam operation and supply the `Uri` values immediately. With `setBeamPushUrisCallback()`, you initiate the beam but do not supply the `Uri` values until the beam connection is established with the peer app.

The `NFC/FileBeam` sample application shows file-based beaming in action.

In our activity (`MainActivity`), in `onCreate()`, we check to make sure that Android Beam is enabled, via a call to `isNdefPushEnabled()` on our `NfcAdapter`. If it is, then we use `ACTION_GET_CONTENT` to retrieve some file from the user (MIME type wildcard of `*/*`):

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  adapter=NfcAdapter.getDefaultAdapter(this);

  if (!adapter.isNdefPushEnabled()) {
    Toast.makeText(this, R.string.sorry, Toast.LENGTH_LONG).show();
    finish();
  }
  else {
    Intent i=new Intent(Intent.ACTION_GET_CONTENT);

    i.setType("*/*").addCategory(Intent.CATEGORY_OPENABLE);
    startActivityForResult(i, 0);
  }
}
```

(from NFC/FileBeam/app/src/main/java/com/commonsware/android/filebeam/MainActivity.java)

In `onActivityResult()`, if we actually got a file (e.g., the result is `ACTION_OK`), we turn around and call `setBeamPushUris()` to pass that file to some peer device. We also set up a `Button` as our UI — clicking the `Button` will `finish()` the activity:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==0 && resultCode==RESULT_OK) {
    adapter.setBeamPushUris(new Uri[] {data.getData()}, this);

    Button btn=new Button(this);

    btn.setText(R.string.over);
    btn.setOnClickListener(this);
    setContentView(btn);
  }
}
```

(from NFC/FileBeam/app/src/main/java/com/commonsware/android/filebeam/MainActivity.java)

That is all there is to it. If you run this app and pick a file, then hold the device up to another Android 4.1+ device, you will be prompted to "Touch to Beam" — doing so will kick off the transfer. Once the transfer is shown on the receiving device, you can pull the devices apart a bit, as the transfer will be proceeding over Bluetooth or WiFi. However, while Bluetooth ranges are much longer than NFC, you still need to keep the devices within a handful of meters of one another.

Note that the receiving device is not running our app. The OS handles the receipt of the transferred file, not our code. Similarly, the OS on the sending device is really the one responsible for the file transfer, so our app does not need the `INTERNET` or `BLUETOOTH` permissions. The downside is that we have no control over anything on the receiving side — the file is stored wherever the OS elects to put it, and the `Notification` it displays when complete will simply launch `ACTION_VIEW` on the pushed file.

Another Sample: SecretAgentMan

To provide another take on using these features of `NfcAdapter`, let's examine the NFC/SecretAgentMan sample application, originally written for a presentation at the 2012 droidcon UK conference. This combines writing to tags, directly beaming text to another device, and using `Uri`-based beaming, all in one app.

The UI of the app is a large `EditText` widget with an action bar:

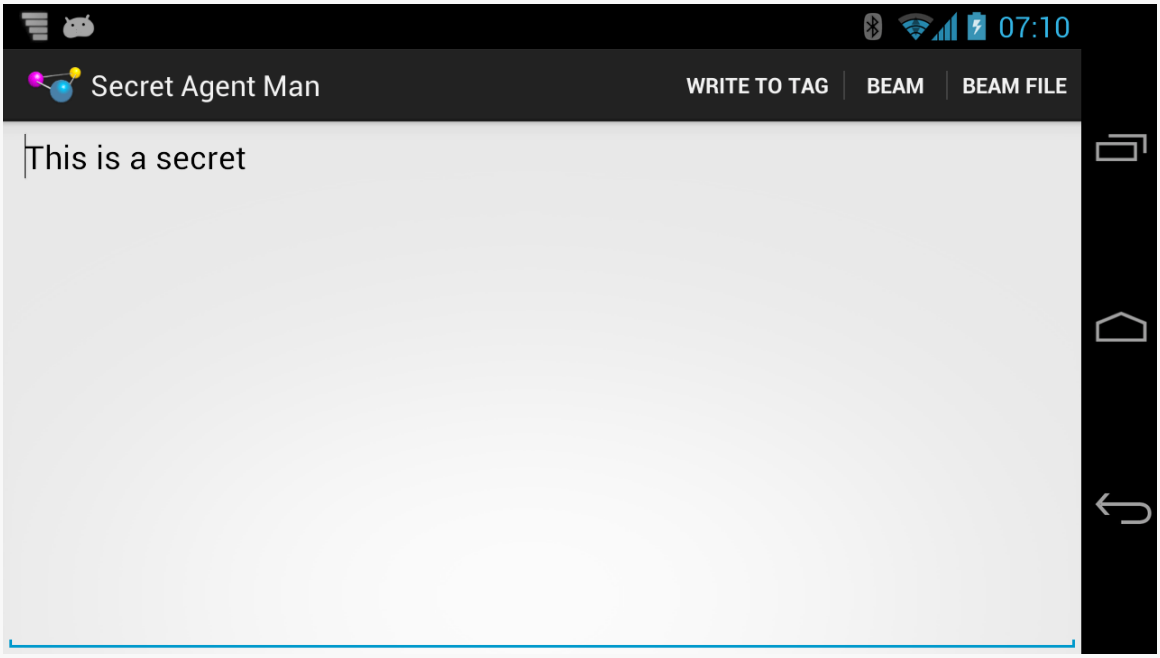


Figure 874: The SecretAgentMan UI

There are three action bar items, one each for the three operations: writing to a tag, directly beaming to another device, and beaming a file (represented via a `Uri`).

Configuration and Initialization

Our app is comprised of a single activity, named `MainActivity`. As part of our manifest setup, we request the `NFC` permission. And, since the app needs NFC to be useful, we also have a `<uses-feature>` element, stipulating that the device needs to have NFC, otherwise the app should not be shown in the Play Store:

```
<uses-permission android:name="android.permission.NFC"/>

<uses-feature
  android:name="android.hardware.nfc"
  android:required="true"/>
```

(from NFC/SecretAgentMan/app/src/main/AndroidManifest.xml)

In `onCreate()` of `MainActivity`, we can then safely get access to an `NfcAdapter`, since the NFC hardware should exist and we have rights to use NFC:

```
@Override
public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    nfc=NfcAdapter.getDefaultAdapter(this);
    secretMessage=(EditText)findViewById(R.id.secretMessage);

    nfc.setOnNdefPushCompleteCallback(this, this);

    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
      readFromTag(getIntent());
    }
  }
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

We also get our hands on the `EditText` widget, storing a reference to it in a data member named `secretMessage`. We will cover the rest of the initialization work in `onCreate()` later in this section, as we cover the code that needs that initialization.

Writing to the Tag

If the user chooses the "Write to Tag" action bar item, we call a `setUpWriteMode()` method from `onOptionsItemSelected()` of `MainActivity`. We maintain an `inWriteMode` `boolean` data member to track whether or not we are already trying to write to an NFC tag. If `inWriteMode` is false, we go ahead and take control over the NFC hardware to attempt to write to the next tag we see:

```
  void setUpWriteMode() {
    if (!inWriteMode) {
      IntentFilter discovery=
          new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
      IntentFilter[] tagFilters=new IntentFilter[] { discovery };
      Intent i=
          new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP
              | Intent.FLAG_ACTIVITY_CLEAR_TOP);
      PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

      inWriteMode=true;
      nfc.enableForegroundDispatch(this, pi, tagFilters, null);
    }
  }
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

To do that, we:

- Create an `IntentFilter` for `ACTION_TAG_DISCOVERED`
- Create a `PendingIntent` for an `Intent` pointing back to this same activity instance (using `getClass()` to identify the instance, plus `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to route control back to our running instance)
- Call `enableForegroundDispatch()` on our `NfcAdapter`, to route newly-discovered tags to us, with the `IntentFilter` identifying the tag-related events we are interested in, and the `PendingIntent` identifying what to do when such a tag is encountered

Once our activity is finishing (e.g., the user presses BACK), we need to clean up our write-to-tag logic. This is kicked off in `onPause()` of `MainActivity`:

```
@Override
public void onPause() {
  if (isFinishing()) {
    cleanUpWritingToTag();
  }

  super.onPause();
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

All we do in `cleanUpWritingToTag()` is discontinue our foreground control over the NFC hardware:

```
void cleanUpWritingToTag() {
  nfc.disableForegroundDispatch(this);
  inWriteMode=false;
}
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

If, before that occurs, the device is tapped on a tag, our activity should regain control in `onNewIntent()` as a result of our `PendingIntent` having been executed:

```
@Override
protected void onNewIntent(Intent i) {
  if (inWriteMode
      && NfcAdapter.ACTION_TAG_DISCOVERED.equals(i.getAction())) {
    writeToTag(i);
  }
  else if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(i.getAction())) {
    readFromTag(i);
```

```
    }
  }
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

If we are in write mode, and if the `Intent` that was just used with `startActivity()` was `ACTION_TAG_DISCOVERED`, we call our `writeToTag()` method to actually start writing information to the tag:

```
  void writeToTag(Intent i) {
    Tag tag=i.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    NdefMessage msg=
        new NdefMessage(new NdefRecord[] { buildNdefRecord() });

    new WriteTagTask(this, msg, tag).execute();
  }
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

To write to the tag, we get our `Tag` out of its `Intent` extra (keyed by `EXTRA_TAG`). Then, we build an `NfcMessage` to write to the tag, getting its `NfcRecord` from `buildNdefRecord()`:

```
  NdefRecord buildNdefRecord() {
    return(new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
                          MIME_TYPE.getBytes(), new byte[] {},
                          secretMessage.getText().toString().getBytes()));
  }
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

Our NDEF record will be of a specific MIME type, represented by a static data member named `MIME_TYPE`:

```
  private static final String MIME_TYPE="vnd.secret/agent.man";
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

The payload of the NDEF record is our "secret message" from the `secretMessage` `EditText` widget.

The `writeToTag()` method then kicks off the same `WriteTagTask` that we used earlier in this chapter:

```
package com.commonsware.android.jimmyb;
```

```
import android.nfc.NdefMessage;
import android.nfc.Tag;
import android.nfc.tech.Ndef;
import android.nfc.tech.NdefFormatable;
import android.os.AsyncTask;
import android.util.Log;
import android.widget.Toast;

class WriteTagTask extends AsyncTask<Void, Void, Void> {
  MainActivity host=null;
  NdefMessage msg=null;
  Tag tag=null;
  String text=null;

  WriteTagTask(MainActivity host, NdefMessage msg, Tag tag) {
    this.host=host;
    this.msg=msg;
    this.tag=tag;
  }

  @Override
  protected Void doInBackground(Void... arg0) {
    int size=msg.toByteArray().length;

    try {
      Ndef ndef=Ndef.get(tag);

      if (ndef == null) {
        NdefFormatable formatable=NdefFormatable.get(tag);

        if (formatable != null) {
          try {
            formatable.connect();

            try {
              formatable.format(msg);
            }
            catch (Exception e) {
              text=host.getString(R.string.tag_refused_to_format);
            }
          }
          catch (Exception e) {
            text=host.getString(R.string.tag_refused_to_connect);
          }
          finally {
            formatable.close();
          }
        }
```

```
        else {
          text=host.getString(R.string.tag_does_not_support_ndef);
        }
      }
      else {
        ndef.connect();

        try {
          if (!ndef.isWritable()) {
            text=host.getString(R.string.tag_is_read_only);
          }
          else if (ndef.getMaxSize() < size) {
            text=host.getString(R.string.message_is_too_big_for_tag);
          }
          else {
            ndef.writeNdefMessage(msg);
            text=host.getString(R.string.success);
          }
        }
        catch (Exception e) {
          text=host.getString(R.string.tag_refused_to_connect);
        }
        finally {
          ndef.close();
        }
      }
    }
    catch (Exception e) {
      Log.e("URLTagger", "Exception when writing tag", e);
      text=host.getString(R.string.general_exception) + e.getMessage();
    }

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    host.cleanUpWritingToTag();

    if (text != null) {
      Toast.makeText(host, text, Toast.LENGTH_SHORT).show();
    }
  }
}
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/WriteTagTask.java)

The net result is that if the user taps the "Write to Tag" action bar item, then taps and holds the device to a tag, we will write a message to the tag and display a `Toast` when we are done.

And, yes, this is a surprising amount of code for what really should be a simple operation...

Reading from the Tag

We can set up `MainActivity` to respond to tags similar to the one we wrote — ones that have the desired MIME Type — via an `android.nfc.action.NDEF_DISCOVERED` `<intent-filter>`:

```
      <intent-filter android:label="@string/app_name">
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <data android:mimeType="vnd.secret/agent.man"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
```

(from NFC/SecretAgentMan/app/src/main/AndroidManifest.xml)

In both `onCreate()` and `onNewIntent()`, if the `Intent` that started our activity is an `NDEF_DISCOVERED` `Intent`, we route control to a `readFromTag()` method:

```
void readFromTag(Intent i) {
  Parcelable[] msgs=
      (Parcelable[])i.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);

  if (msgs.length > 0) {
    NdefMessage msg=(NdefMessage)msgs[0];

    if (msg.getRecords().length > 0) {
      NdefRecord rec=msg.getRecords()[0];

      secretMessage.setText(new String(rec.getPayload(), US_ASCII));
    }
  }
}
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

In principle, there could be several NDEF messages on the tag, but we only pay attention to the first element, if any, of the `EXTRA_NDEF_MESSAGES` array of `Parcelable` objects on the `Intent`. Similarly, in principle, there could be several NDEF records in the first message, but we only examine the first element out of the

array of `NdefRecord` objects contained in the `NdefMessage`. From there, we extract our secret message and display it by means of putting it in the `EditText` widget.

Beaming the Text

This sample only supports beaming — whether of NDEF messages directly or of a file — if we are on API Level 16 or higher. Hence, in `onCreateOptionsMenu()`, we check our version and only enable our default-disabled beam action bar items if:

- We are on API Level 16 or higher, and
- NDEF push mode is enabled, via a call to `isNdefPushEnabled()` on our `NfcAdapter`:

```
@TargetApi(16)
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.activity_main, menu);

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
    menu.findItem(R.id.simple_beam)
        .setEnabled(nfc.isNdefPushEnabled());
    menu.findItem(R.id.file_beam).setEnabled(nfc.isNdefPushEnabled());
  }

  return(super.onCreateOptionsMenu(menu));
}
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

If the user taps on the "Beam" action bar item, we call an `enablePush()` method from `onOptionsItemSelected()`, which simply enables push mode:

```
void enablePush() {
  nfc.setNdefPushMessageCallback(this, this);
}
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

We arrange for the activity itself to be the `CreateNdefMessageCallback` necessary for push mode. That requires us to implement `createNdefMessage()`, which will be called if we are in push mode and a push-compliant device comes within range:

```
@Override
public NdefMessage createNdefMessage(NfcEvent event) {
  return(new NdefMessage(
                       new NdefRecord[] {
```

```
                            buildNdefRecord(),
                            NdefRecord.createApplicationRecord("com.commonsware.android.jimmyb") }));
  }
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

Here, we create an `NdefMessage` similar to the one we wrote to the tag earlier in this sample. However, we also attach an Android Application Record (AAR), by means of the static `createApplicationRecord()` method on `NdefRecord`. This, in theory, will help route the push to our app on the other device, including downloading it from the Play Store if needed (and, of course, if it actually existed on the Play Store, which it does not).

Back up in `onCreate()`, we call `setOnNdefPushCompleteCallback()`, to be notified of when a push operation is completed. Once again, we set up `MainActivity` to be the callback, this time by implementing the `OnNdefPushCompleteCallback` interface. That, in turn, requires us to implement `onNdefPushComplete()`, where we disable push mode via a call to `setNdefPushMessageCallback()` with a `null` listener:

```
@Override
public void onNdefPushComplete(NfcEvent event) {
  nfc.setNdefPushMessageCallback(null, this);
}
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

To receive the beam, we only need our existing logic to read from the tag, as on the receiving side, a push is indistinguishable from reading a tag, and we are using the same MIME type for both the message written to the tag and the message we are pushing.

Beaming the File

If the user taps the "Beam File" action bar item, we find some file to beam, by means of an `ACTION_GET_CONTENT` request and `startActivityForResult()`:

```
    case R.id.file_beam:
      Intent i=new Intent(Intent.ACTION_GET_CONTENT);

      i.setType("*/*").addCategory(Intent.CATEGORY_OPENABLE);
      startActivityForResult(i, 0);
      return(true);
```

(from NFC/SecretAgentMan/app/src/main/java/com/commonsware/android/jimmyb/MainActivity.java)

In `onActivityResult()`, if the request succeeded, we use `setBeamPushUris()` to tell Android to beam the selected file to another device. Nothing more is needed on our side, and the receipt of the file is handled entirely by the OS, not our application code, so there is nothing to be written for that.

This code assumes the NFC adapter is enabled. We could check that via a call to `isEnabled()` on our `NfcAdapter`. If it is not enabled, we could — on user request — bring up the Settings activity for configuring NFC, via `startActivity(new Intent(Settings.ACTION_NFC_SETTINGS))`. However, oddly, this `Intent` action is only available on Android 4.1 (API Level 16) and higher, despite NFC having been available for some time previously.

This code ignores the possibility of doing the simple beam (not the file-based beam) on Android 4.0.x devices. That is because the `isNdefPushEnabled()` method was not added until Android 4.1, and therefore we do not know whether or not we can actually do a beam.

If `isNdefPushEnabled()` returns `false`, we simply disable some action bar items. Alternatively, we could use `startActivity(new Intent(Settings.ACTION_NFCSHARING_SETTINGS))`, on API Level 14 and higher, to bring up the beam screen in Settings, to allow the user to toggle beam support on.

Additional Resources

To help make sense of the tags that you are trying to use with your app, you may wish to grab the NFC TagInfo application off of the Google Play Store. This application simply scans a tag and allows you to peruse all the details of that tag, including the supported technologies (e.g., does it support NDEF? is it `NdefFormatable`?), the NDEF records, and so on.

To learn more about NFC on Android — beyond this chapter or the Android developer documentation – this Google I|O 2011 presentation is recommended.

Device Administration

Balding authors of Android books often point out that enterprises and malware authors have the same interests: they want to take control of a device away from the person that is holding it and give that control to some other party. Android, being a consumer operating system, is designed to defend against malware, and so enterprises can run into issues.

However, Android does have a growing area of device administration APIs, that allow carefully-constructed and installed applications to exert some degree of control over the device, how it is configured, and how it operates.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on broadcast `Intents`.

Objectives and Scope

One might read the phrase "device administration" and assume that somebody, using these APIs, could do anything they want on the device.

That's not quite what "device administration" means in this case.

Rather, the device administration APIs serve three main roles:

1. They allow an application to dictate how well a device is secured, from the password required in the OS lock screen to whether the device should have full-disk encryption

2. They allow an application to find out when security issues might arise, notably failed password attempts
3. They allow an application to lock the device, disable its cameras, or even perform a "wipe" (i.e., factory reset)

The user, however, has to agree to enable a device administration app. It does not magically get all these powers simply by being installed. What the user gets from agreeing to this is access to something that otherwise would be denied (e.g., to use Enterprise App X, you must agree to allow it to be a device administrator).

Defining and Registering an Admin Component

There are four pieces for defining and registering a device administration app: creating the metadata, adding the `<receiver>` to the manifest, implementing that `BroadcastReceiver`, and telling Android to ask the user to agree to allow the app to a device administrator.

Here, we will take a peek at the `DeviceAdmin/LockMeNow` sample application.

The Feature

Apps implementing device administrators should add a `<uses-feature>` element with a name of `android.software.device_admin`, indicating whether or not they require this device feature to exist. This can be used by the Play Store to filter your app from being available on devices that, for one reason or another, do not offer this capability.

The Metadata

As with app widgets and other Android facilities, you will need to define a metadata file as an XML resource, describing in greater detail what your device administration app wishes to do. This information will determine what you will be allowed to do once the user approves your app, and what you list here will be displayed to the user when you request such approval.

The `DeviceAdminInfo` class has a series of static data members (e.g., `USES_ENCRYPTED_STORAGE`) that represent specific policies that your device administrator app could use. The documentation for each of those static data members lists the corresponding element that goes in this XML metadata file (e.g., `<encrypted-storage>`). These elements are wrapped in a `<uses-policies>` element,

which itself is wrapped in a `<device-admin>` element. The range of possible policies is shown in the following sample XML metadata file:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-policies>
    <disable-camera />
    <encrypted-storage />
    <expire-password />
    <force-lock />
    <limit-password />
    <reset-password />
    <watch-login />
    <wipe-data />
  </uses-policies>
</device-admin>
```

Here, we:

- Intend to disable the cameras, if needed
- Will ask the user to encrypt their device storage, if it has not been done already
- Will set an expiration time for the user's password, after which they will need to set up a new one
- Intend to lock the device, if needed
- Will set criteria for password quality, such as minimum length
- Intend to forcibly reset the user's password, if needed
- Intend to monitor for failed and successful login attempts
- Intend to wipe the device, if needed

Choose which of those policies you need — the fewer you request, the more likely it is the user will not wonder about your intentions. In your project's `res/xml/` directory, create a file that looks like the above with the policies you wish. You can name this file whatever you want (e.g., `device_admin.xml`), within standard Android resource naming rules.

The Manifest

In the manifest, you will need to declare a `<receiver>` element for the `DeviceAdminReceiver` component that you will write. This component not only is the embodiment of the device admin capabilities of your app, but it will be the one notified of failed logins and other events.

For example, here is the `<receiver>` element from the `LockMeNow` sample app:

```
    <receiver
      android:name="AdminReceiver"
      android:permission="android.permission.BIND_DEVICE_ADMIN">
      <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin"/>

      <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
      </intent-filter>
    </receiver>
```

(from DeviceAdmin/LockMeNow/app/src/main/AndroidManifest.xml)

There are three things distinctive about this element compared to your usual `<receiver>` element:

1. It requires that whoever sends broadcasts to it hold the `BIND_DEVICE_ADMIN` permission. Since that permission is protected and can only be held by apps signed with the firmware's signing key, you can be reasonably assured that any events sent to you are real.
2. It has the `<meta-data>` child element pointing to our device administration metadata from the previous section.
3. It registers for `android.app.action.DEVICE_ADMIN_ENABLED` broadcasts via its `<intent-filter>` — this is the broadcast that will be used to notify you about failed logins or other events.

The Receiver

The `DeviceAdminReceiver` itself needs to exist as a component in your app, registered in the manifest as shown above. At minimum, though, it does not need to override any methods, such as the implementation from the `LockMeNow` sample app:

```
package com.commonsware.android.lockme;

import android.app.admin.DeviceAdminReceiver;

public class AdminReceiver extends DeviceAdminReceiver {
}
```

(from DeviceAdmin/LockMeNow/app/src/main/java/com/commonsware/android/lockme/AdminReceiver.java)

By requesting the `DEVICE_ADMIN_ENABLED` broadcasts, we could get control when we are enabled by overriding an `onEnabled()` method. We could also register for other

broadcasts (e.g., `ACTION_PASSWORD_FAILED`) and implement the corresponding callback method on our `DeviceAdminReceiver` (e.g., `onPasswordFailed()`).

The Demand for Device Domination

Simply having this component in our manifest, though, is insufficient. The user must proactively agree to allow us to administer their device. And, since this is potentially very dangerous, a simple permission was deemed to also be insufficient. Instead, we need to ask the user to approve us as a device administrator from our app, typically from an activity.

In the case of `LockMeNow`, the UI is just a really big button, tied to a `lockMeNow()` method on our `LockMeNowActivity`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/Button1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:onClick="lockMeNow"
    android:text="@string/lock_me"
    android:textColor="#FFFF0000"
    android:textSize="40sp"
    android:textStyle="bold"/>

</LinearLayout>
```

(from DeviceAdmin/LockMeNow/app/src/main/res/layout/main.xml)

In `onCreate()` of the activity, in addition to loading up the UI via `setContentView()`, we create a `ComponentName` object identifying our `AdminReceiver` component. We also request access to the `DevicePolicyManager`, via a call to `getSystemService()`. `DevicePolicyManager` is our gateway for making direct requests for device administration operations, such as locking the device:

```
package com.commonsware.android.lockme;

import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
```

```
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class LockMeNowActivity extends Activity {
  private DevicePolicyManager mgr=null;
  private ComponentName cn=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
    cn=new ComponentName(this, AdminReceiver.class);
    mgr=(DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);
  }

  public void lockMeNow(View v) {
    if (mgr.isAdminActive(cn)) {
      mgr.lockNow();
    }
    else {
      Intent intent=
          new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
      intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, cn);
      intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                      getString(R.string.device_admin_explanation));
      startActivity(intent);
    }
  }
}
```

(from DeviceAdmin/LockMeNow/app/src/main/java/com/commonsware/android/lockme/LockMeNowActivity.java)

In `lockMeNow()`, we ask the `DevicePolicyManager` if we have already been registered as a device administrator, by calling `isAdminActive()`, supplying the `ComponentName` of our `DeviceAdminReceiver` that should be so registered. If that returns `false`, then the user has not approved us as a device administrator yet, so we need to ask them to do so. To do that, you:

- Create an `Intent` for the `DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN` action
- Add the `ComponentName` of our `DeviceAdminReceiver` as an extra, keyed as `DevicePolicyManager.EXTRA_DEVICE_ADMIN`

- Add another extra, `DevicePolicyManager.EXTRA_ADD_EXPLANATION`, which is some text to show the user as part of the authorization screen, to explain why we need to be a device admin
- Start up an activity using that `Intent`, via `startActivity()`

If you run this on a device, then tap the button, the first time you do so the user will be prompted to agree to making the app be a device administrator:

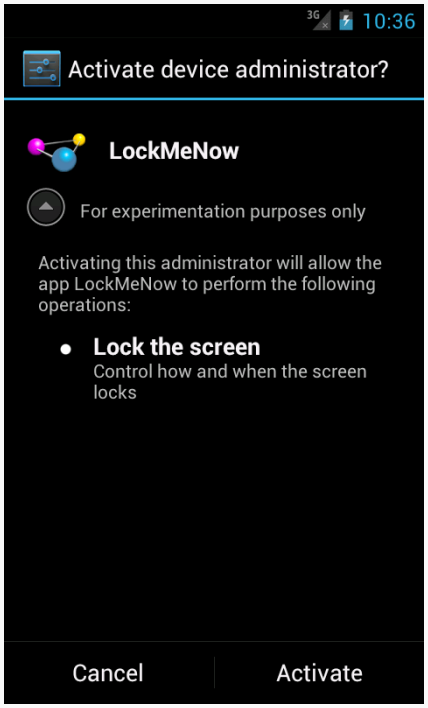


Figure 875: The Activate Device Administrator Screen

The "For experimentation purposes only" is the value of our `DevicePolicyManager.EXTRA_ADD_EXPLANATION` extra, loaded from a string resource.

If the user clicks "Activate", and you overrode `onEnabled()` in your `DeviceAdminReceiver`, that will be called to let you know that you have been approved and can perform device administration functions. Your component will also appear in the list of device administrators in the Settings app:

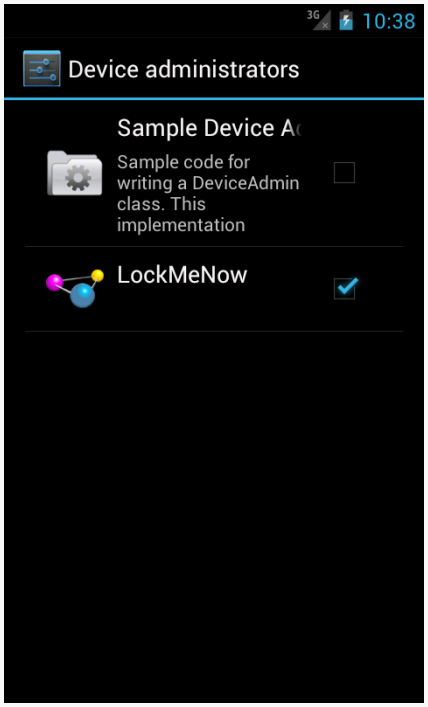


Figure 876: The Device Administrator List

The user can, at any time, uncheck you in this list and disable you. You can find out about this by having your `DeviceAdminReceiver` listen for `ACTION_DEVICE_ADMIN_DISABLE_REQUESTED` broadcasts and overriding the `onDisableRequested()` method, where you can return the text of a message to be displayed to the user confirming that they do indeed wish to go ahead with the disable operation. To find out if they go through with it, your `DeviceAdminReceiver` can listen for `ACTION_DEVICE_ADMIN_DISABLED` broadcasts and override `onDisabled()`.

Going Into Lockdown

Given that the user has approved your device administration request, and given that you requested `<force-lock>` in your metadata, you can call `lockNow()` on a `DevicePolicyManager`. That will immediately lock the device and (generally) turn off the screen. It is as if the user pressed the POWER button on the device. If anything, `lockNow()` will offer tighter security.

The `LockItNow` sample app does this if, when the user clicks the really big button, it detects that it is already a device administrator. If you test this on a device, it will

behave as though the user pressed POWER; on an emulator, you will need to press the HOME button to "power on" the screen and be able to re-enter your emulator.

You can also call:

- `setCameraDisabled()` to disable all cameras, if you requested `<disable-camera>` in the metadata. Note that this disables all cameras; there is no provision at this time to disable individual cameras separately.
- `wipeData()`, which performs what amounts to a factory reset — it leaves external storage alone but wipes the contents of internal storage as part of a reboot. This requires the `<wipe-data>` policy in the metadata.
- `setKeyguardDisabledFeatures()`, to control whether or not the lockscreen allows direct access to the camera and/or app widgets (lockscreen app widgets are described in the chapter on app widgets)

For example, the latter feature, while available in the Android SDK, is not built into the Settings app of Android 4.2. As a result, users need a third-party app to toggle on or off lockscreen access to the camera and app widgets. One such third-party app is LockscreenLocker, released as open source by the author of this book.

Basically, the app presents you with two `Switch` widgets to control the camera and app widgets on the lock screen. First, though, it shows you a message and a `Button`, if the app is not set up as a device administrator:

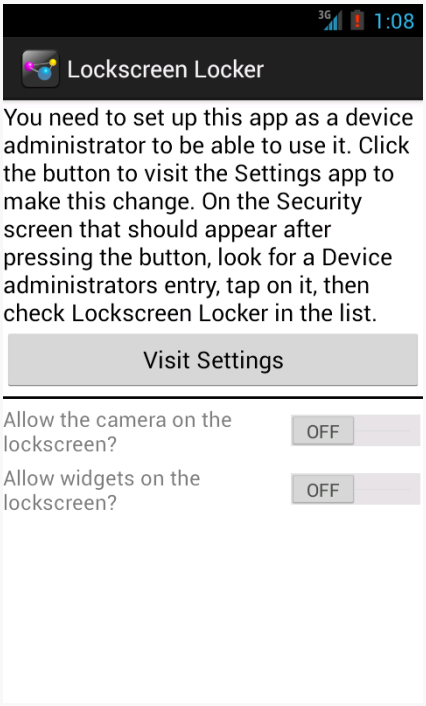


Figure 877: LockscreenLocker, On Initial Run

Once that is complete, the `Switch` widgets become enabled and usable:

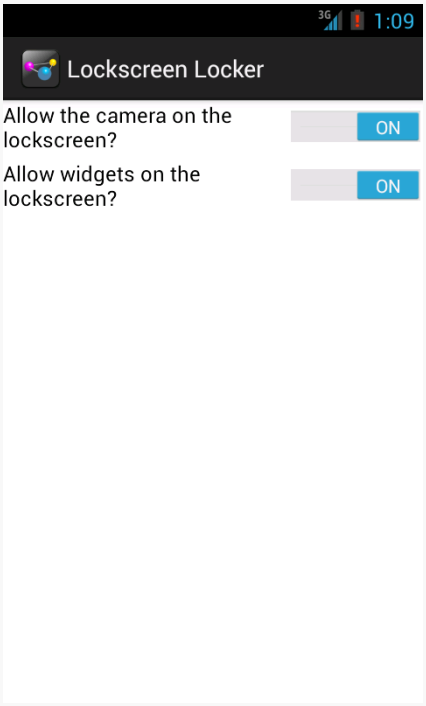


Figure 878: LockscreenLocker, After Being Made a Device Admin

The device admin metadata for this app specifies that we want to control keyguard features:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-policies>
    <disable-keyguard-features/>
  </uses-policies>

</device-admin>
```

Note that, at the time of this writing, there is a flaw in the Android developer documentation — the correct element to have in the metadata is `<disable-keyguard-features/>`, not `<disable-keyguard-widgets>`. You can track this issue to see when this documentation bug has been repaired.

Our device admin component, `LockscreenAdminReceiver`, is empty, because there are no events that we are trying to listen to:

```
public class LockscreenAdminReceiver extends DeviceAdminReceiver {
}
```

However, we still need the `LockscreenAdminReceiver`, as it is the component that is tied to our device admin metadata and indicates to the system that we should be an option in Settings for available device administrators.

Our activity layout contains all the requisite widgets: a `TextView` for the message, a `Button` to jump to the Settings app, a `View` to serve as a divider, and a pair of `Switch` widgets to manage the lockscreen settings:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:id="@+id/setupMessage"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/setup_message"
    android:textAppearance="?android:attr/textAppearanceMedium"
    android:visibility="gone"/>

  <Button
    android:id="@+id/setup"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showSettings"
    android:text="@string/visit_settings"
    android:visibility="gone"/>

  <View
    android:id="@+id/divider"
    android:layout_width="match_parent"
    android:layout_height="2dip"
    android:layout_marginBottom="4dip"
    android:layout_marginTop="4dip"
    android:background="#FF000000"
    android:visibility="gone"/>

  <Switch
    android:id="@+id/camera"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/allow_camera"/>

  <Switch
```

```
    android:id="@+id/widgets"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="4dip"
    android:text="@string/allow_widgets"/>

</LinearLayout>
```

In `onCreate()` of our activity (`MainActivity`), we request a `DevicePolicyManager`, set up a `ComponentName` identifying our `DeviceAdminReceiver` implementation (`LockscreenAdminReceiver`), and hook up the activity to know about changes in the state of the `Switch` widgets:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mgr=(DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);
    cn=new ComponentName(this, LockscreenAdminReceiver.class);

    camera=(CompoundButton)findViewById(R.id.camera);
    camera.setOnCheckedChangeListener(this);

    widgets=(CompoundButton)findViewById(R.id.widgets);
    widgets.setOnCheckedChangeListener(this);
  }
```

In `onResume()`, we check to see if our `DeviceAdminReceiver` is active — in other words, whether the user has set us up as being a device administrator or not:

```
  @Override
  public void onResume() {
    super.onResume();

    if (mgr.isAdminActive(cn)) {
      toggleWidgets(true);

      int status=mgr.getKeyguardDisabledFeatures(cn);

      camera.setChecked(!((status & DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA) ==
DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA));
      widgets.setChecked(!((status & DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL) ==
DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL));
    }
    else {
      toggleWidgets(false);
    }
  }
```

We toggle the visibility and enabled settings of our widgets based upon whether we are a device administrator or not, in a `toggleWidgets()` private method:

```
private void toggleWidgets(boolean enable) {
  int visibility=(enable ? View.GONE : View.VISIBLE);

  camera.setEnabled(enable);
  widgets.setEnabled(enable);

  findViewById(R.id.divider).setVisibility(visibility);
  findViewById(R.id.setup).setVisibility(visibility);
  findViewById(R.id.setupMessage).setVisibility(visibility);
}
```

`onResume()` also sets the state of our `Switch` widgets based upon the current state of the keyguard features, by calling `getKeyguardDisabledFeatures()` on the `DevicePolicyManager`. This returns a bit set of which features are disabled, with `DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA` and/or `DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL` possibly being set.

At the outset, after being installed, we will not be a device administrator, so the `Switch` widgets will be disabled and the `Button` will be visible. We simply send the user to the security screen in the Settings app if they click that button:

```
public void showSettings(View v) {
  startActivity(new Intent(Settings.ACTION_SECURITY_SETTINGS));
}
```

When the user toggles a `Switch`, our activity will be called with `onCheckedChanged()`. There, we need to call `setKeyguardDisabledFeatures()` with a new bit set, toggling on or off a bit based on the user's chosen values in the UI:

```
@Override
public void onCheckedChanged(CompoundButton buttonView,
                             boolean isChecked) {
  int status=mgr.getKeyguardDisabledFeatures(cn);

  if (buttonView == camera) {
    if (isChecked) {
      mgr.setKeyguardDisabledFeatures(cn, status
          & ~DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA);
    }
    else {
      mgr.setKeyguardDisabledFeatures(cn, status
          | DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA);
```

```
      }
    }
    else {
      if (isChecked) {
        mgr.setKeyguardDisabledFeatures(cn, status
            & ~DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL);
      }
      else {
        mgr.setKeyguardDisabledFeatures(cn, status
            | DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL);
      }
    }
  }
```

Note that we have the `Switch` widgets set up for positive statements (e.g., "enable the camera"), while the bit set uses negative statements (e.g., "disable the camera"). That makes toggling the bit set a "bit" more complicated, to ensure that we are applying the user's choices correctly.

Passwords and Device Administration

One popular facet of the device administration APIs is for an app to mandate a certain degree of password quality. The app might then fail to operate if the current password does not meet the requested quality standard.

Mandating Quality of Security

You can call various setters on `DevicePolicyManager` to dictate your minimum requirements for the password that the user uses to get past the lock screen. Examples include:

- `setPasswordMinimumLength()`
- `setPasswordQuality()` (with an integer flag describing the type of "quality" you seek, such as `PASSWORD_QUALITY_NUMERIC` if a PIN is OK, or `PASSWORD_QUALITY_COMPLEX` if you require mixed case and numbers and such)
- `setPasswordMinimumLowerCase()` (indicating how many lowercase letters are required at minimum in the user's password)

All of these require the `<limit-password>` policy be requested in the metadata.

Then, you can call `isActivePasswordSufficient()` to determine if the current password meets your requirements. If it does not, you might elect to disable certain functionality. Or, if you requested the `<reset-password>` policy in the metadata, you can call `resetPassword()` to force the user to come up with a password meeting your requirements.

Similarly, you can also call `getStorageEncryptionStatus()` on `DevicePolicyManager` to find out whether full-disk encryption is active, inactive, or unavailable on this particular device. If it is inactive, and you requested the `<encrypted-storage>` policy in your metadata, you can call `setStorageEncryption()` to demand it, and start the encryption process via starting the `ACTION_START_ENCRYPTION` activity.

Establishing Password Requirements

To see password quality enforcement in action, let us examine the DeviceAdmin/PasswordEnforcer sample application.

The activity (`MainActivity`) is fairly short, and much of its code is based on the earlier `LockMeNow` sample:

```
package com.commonsware.android.pwenforce;

import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;
import android.widget.Toast;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ComponentName cn=new ComponentName(this, AdminReceiver.class);
    DevicePolicyManager mgr=
        (DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);

    if (mgr.isAdminActive(cn)) {
      int msgId;

      if (mgr.isActivePasswordSufficient()) {
        msgId=R.string.compliant;
```

```
      }
      else {
        msgId=R.string.not_compliant;
      }

      Toast.makeText(this, msgId, Toast.LENGTH_LONG).show();
    }
    else {
      Intent intent=
          new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
      intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, cn);
      intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                      getString(R.string.device_admin_explanation));
      startActivity(intent);
    }

    finish();
  }
}
```

(from DeviceAdmin/PasswordEnforcer/app/src/main/java/com/commonsware/android/pwenforce/MainActivity.java)

In `onCreate()`, after obtaining a `DevicePolicyManager`, we see if our app has been designated by the user as a device administrator. If not — which will be the case when the app is first installed — we use an `ACTION_ADD_DEVICE_ADMIN` `Intent` and `startActivity()` to steer the user towards making our app be a device administrator.

If the user does make our app be a device administrator, our `AdminReceiver` will get control in `onEnabled()`, as we have registered it for `DEVICE_ADMIN_ENABLED` broadcasts in the manifest. In `onEnabled()`, we mandate that the password for the device must be alphanumeric, via a call to `setPasswordQuality()` on the `DevicePolicyManager`:

```
@Override
public void onEnabled(Context ctxt, Intent intent) {
  ComponentName cn=new ComponentName(ctxt, AdminReceiver.class);
  DevicePolicyManager mgr=
      (DevicePolicyManager)ctxt.getSystemService(Context.DEVICE_POLICY_SERVICE);

  mgr.setPasswordQuality(cn,
                         DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC);

  onPasswordChanged(ctxt, intent);
}
```

(from DeviceAdmin/PasswordEnforcer/app/src/main/java/com/commonsware/android/pwenforce/AdminReceiver.java)

We will see the role of the `onPasswordChanged()` method, called late in `onEnabled()`, later in this chapter.

Back in `onCreate()` of our `MainActivity`, if we *are* a device administrator, then we know that the `setPasswordQuality()` call has been made, and so we can check to see if the current password meets our standards via a call to `isActivePasswordSufficient()` on the `DevicePolicyManager`. The app displays a `Toast` showing whether the password is or is not currently "sufficient".

Password-Related Events

Via appropriate actions in our `<intent-filter>` for our `DeviceAdminReceiver`, and associated callback methods, we can find out other things that go on with respect to the password:

- `ACTION_PASSWORD_CHANGED` informs us when the user has changed her password
- `ACTION_PASSWORD_FAILED` informs us when somebody tries to enter a password, and the password was incorrect
- `ACTION_PASSWORD_SUCCEEDED` informs us when the user has successfully entered the password and unlocked the device... after an attempt had previously failed

The `PasswordEnforcer` sample registers for all of these in the manifest:

```
    <receiver
      android:name="AdminReceiver"
      android:permission="android.permission.BIND_DEVICE_ADMIN">
      <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin"/>

      <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
        <action android:name="android.app.action.ACTION_PASSWORD_CHANGED"/>
        <action android:name="android.app.action.ACTION_PASSWORD_FAILED"/>
        <action android:name="android.app.action.ACTION_PASSWORD_SUCCEEDED"/>
      </intent-filter>
    </receiver>
```

(from DeviceAdmin/PasswordEnforcer/app/src/main/AndroidManifest.xml)

The implementations of the corresponding `onPasswordChanged()`, `onPasswordFailed()`, and `onPasswordSucceeded()` methods simply display `Toast` messages about those events:

```
@Override
public void onPasswordChanged(Context ctxt, Intent intent) {
  DevicePolicyManager mgr=
      (DevicePolicyManager)ctxt.getSystemService(Context.DEVICE_POLICY_SERVICE);
  int msgId;

  if (mgr.isActivePasswordSufficient()) {
    msgId=R.string.compliant;
  }
  else {
    msgId=R.string.not_compliant;
  }

  Toast.makeText(ctxt, msgId, Toast.LENGTH_LONG).show();
}

@Override
public void onPasswordFailed(Context ctxt, Intent intent) {
  Toast.makeText(ctxt, R.string.password_failed, Toast.LENGTH_LONG)
       .show();
}

@Override
public void onPasswordSucceeded(Context ctxt, Intent intent) {
  Toast.makeText(ctxt, R.string.password_success, Toast.LENGTH_LONG)
       .show();
}
```

(from DeviceAdmin/PasswordEnforcer/app/src/main/java/com/commonsware/android/pwenforce/AdminReceiver.java)

However, these will illustrate some quirks in the behavior of the device administration APIs:

- `onPasswordSucceeded()` is not called on *every* successful password entry, only those that come after a prior `onPasswordFailed()` call. One imagines that perhaps `onPasswordSucceededAfterItHadFailedBefore()` was deemed to be too wordy.
- `isActivePasswordSufficient()` will return a value based on the *previous* password in `onPasswordChanged()`, not the newly-changed password. Since the system will prevent the user from entering a new password that is insufficient, you should not need to call `isActivePasswordSufficient()` from `onPasswordChanged()`.
- A `Toast` cannot display over the lockscreen, and so the `onPasswordFailed()` `Toast` will never be seen.

Getting Along with Others

Bear in mind that you might not be the only device administrator on any given device. If there are multiple administrators, the most secure requirements are in force. So, for example, if Admin A requests a minimum password length of 7, and Admin B requests a minimum password length of 10, the user will have to supply a password that is at least 10 characters long, to meet both device administrators' requirements.

This also means that certain requests you make may fail. For example, if you decide to say that you do *not* need encryption (`setStorageEncryption()` with a value of `false`), if something *else* needs encryption, the user will still need to encrypt their device.

Basic Use of Sensors

"Sensors" is Android's overall term for ways that Android can detect elements of the physical world around it, from magnetic flux to the movement of the device. Not all devices will have all possible sensors, and other sensors are likely to be added over time. In this chapter, we will explore the general concept of Android sensors and how to receive data from them.

Note, however, that this chapter will not get into details of detecting movement via the accelerometer, etc.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on threads. Having experience with other system-service-and-listener patterns, such as fetching locations with `LocationManager`, is helpful but not strictly required.

The Sensor Abstraction Model

When fetching locations from `LocationManager`, you do not have dedicated APIs per location-finding technology (e.g., GPS vs. WiFi hotspot proximity vs. cell-tower triangulation vs. ...). Instead, you work with a `LocationManager` system service, asking for locations using a single API, where location technologies are identified by name (e.g., `GPS_PROVIDER`).

Similarly, when working with sensors, you do not have dedicated APIs to get sensor readings from each sensor. Instead, you work with a `SensorManager` system service,

asking for sensor events using a single API, where sensors are identified by name (e.g., `TYPE_LINEAR_ACCELERATION`).

Note, though, that there are some dedicated methods on `SensorManager` to help you *interpret* some of the sensors, particularly the accelerometer. However, those are merely helper methods; getting at the actual accelerometer data uses the same APIs that you would use to, say, access the barometer for atmospheric pressure.

Considering Rates

Usually, when working with sensors, you want to find out about changes in the sensor reading over a period of time. For example, in a driving game, where the user holds their device like a steering wheel and uses it to "turn" their virtual car, you need to know information about acceleration and positioning so long as game play is going on.

Hence, when you request a feed of sensor readings from `SensorManager`, you will specify a desired rate at which you should receive those readings. You do that by specifying an amount of delay in between readings; Android will drop sensor readings that arrive before the delay period has elapsed.

There are four standard delay periods, defined as constants on the `SensorManager` class:

1. `SENSOR_DELAY_NORMAL`, which is what most apps would use for broad changes, such as detecting a screen rotating from portrait to landscape
2. `SENSOR_DELAY_UI`, for non-game cases where you want to update the UI continuously based upon sensor readings
3. `SENSOR_DELAY_GAME`, which is faster (less delay) than `SENSOR_DELAY_UI`, to try to drive a higher frame rate
4. `SENSOR_DELAY_FASTEST`, which is the "firehose" of sensor readings, without delay

The more sensor readings you get, the faster your code has to be for using those readings, lest you take too long and starve your thread of time to do anything else. This is particularly important given that you receive these sensor events on the main application thread, and therefore the time you spend processing these events is time unavailable for screen updates. Hence, choose the slowest rate that you can that will give you acceptable granularity of output.

Reading Sensors

Sensors are event-driven. You cannot ask Android for the value of a sensor at a point in time. Rather, you register a listener for a sensor, then process the sensor events as they come in. You can unregister the listener when you are done, either because you have the reading that you need, or the user has done something (like move to another activity) that indicates that you no longer need the sensor events.

To demonstrate this, we will examine the `Sensor/Monitor` sample application, which will list all of the available sensors, plus show the incoming readings from a selected sensor.

Obtaining a SensorManager

The gateway to the sensor roster on the device is the `SensorManager` system service. You obtain one of these by calling `getSystemService()` on any `Context`, asking for the `SENSOR_SERVICE`, and casting the result to be a `SensorManager`, as seen in the `onCreate()` method of our `MainActivity`:

```
    mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/MainActivity.java)

Identifying a Sensor of Interest

There are sensor types, and then there are sensors.

You might think that there would be a one-to-one mapping between these. In truth, there might be more than one sensor for a given type, the way the `SensorManager` API is set up. Regardless, somewhere along the line, you will need to identify the `Sensor` that you want to work with.

The most common pattern, if you know the type of sensor that you want, is to call `getDefaultSensor()` on `SensorManager`, supplying the type of the sensor (e.g., `TYPE_ACCELEROMETER`, `TYPE_GYROSCOPE`), where the type names are constants defined on the `Sensor` class. If there is more than one possible `Sensor` for that type, Android will give you the "default" one, which is usually a reasonable choice.

Another approach, and the one used by this sample application, is to call `getSensorList()` on `SensorManager`, which returns a `List` of all `Sensor` objects

available on this device. The sample's `MainActivity` has a `getSensorList()` that returns this list, after a bit of manipulation:

```
@Override
public List<Sensor> getSensorList() {
  List<Sensor> unfiltered=
      new ArrayList<Sensor>(mgr.getSensorList(Sensor.TYPE_ALL));
  List<Sensor> result=new ArrayList<Sensor>();

  for (Sensor s : unfiltered) {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT
        || !isTriggerSensor(s)) {
      result.add(s);
    }
  }

  Collections.sort(result, new Comparator<Sensor>() {
    @Override
    public int compare(final Sensor a, final Sensor b) {
      return(a.toString().compareTo(b.toString()));
    }
  });

  return(result);
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/MainActivity.java)

Android 4.4 started introducing some "trigger sensors", ones that are designed to deliver a single reading, then automatically become unregistered. This sample app is designed to display results from more traditional sensors that provide ongoing readings. So, `getSensorList()` calls an `isTriggerSensor()` method on API Level 19+ devices, and throws out sensors that are trigger sensors. The `isTriggerSensor()` method simply checks the sensor type against a list of trigger sensors:

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private boolean isTriggerSensor(Sensor s) {
  int[] triggers=
      { Sensor.TYPE_SIGNIFICANT_MOTION, Sensor.TYPE_STEP_DETECTOR,
          Sensor.TYPE_STEP_COUNTER };

  return(Arrays.binarySearch(triggers, s.getType()) >= 0);
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/MainActivity.java)

The reason for isolating `isTriggerSensor()` into a separate method, and not having the array of sensor types as a `static final` array, is because these sensor types are not available in all Android versions. Having the array of sensor types as a `static final` data member would require putting the `@TargetApi` annotation on the entire class, which is unwise if the class will be used on older devices. This way, we can isolate the new-target code into a dedicated method, with a more locally-scoped `@TargetApi` annotation.

Getting Sensor Events

To get sensor events, you need a `SensorEventListener`. This is an interface, calling for two method implementations:

1. `onAccuracyChanged()`, where you are informed about a significant change in the accuracy of the readings that you are going to get from the sensor
2. `onSensorChanged()`, where you are passed a `SensorEvent` representing one of those readings

To receive events for a given `Sensor`, you call `registerListener()` on the `SensorManager`, supplying the `Sensor`, the `SensorEventListener`, and one of the `SENSOR_DELAY_*` values to control the rate of events. Later on, you need to call `unregisterListener()`, supplying the same `SensorEventListener`, to break the connection. **Failing to unregister the listener is bad**. The sensor subsystem is oblivious to things like activity lifecycles, and so if you leak a listener, not only will you perhaps leak the component that registered the listener, but you will continue to get sensor events until the process is terminated. As active sensors do consume power, users will not appreciate the battery drain your leaked listener will incur.

The `List` of `Sensor` objects from that `getSensorList()` method shown previously will be used to populate a `ListView`. When the user taps on a `Sensor` in the list, an `onSensorSelected()` method is called on the `MainActivity`. Here, we unregister our listener (a `SensorLogFragment` that we will discuss more in a bit), in case we were registered for a prior `Sensor` choice, before registering for the newly-selected `Sensor`:

```
@Override
public void onSensorSelected(Sensor s) {
  mgr.unregisterListener(log);
  mgr.registerListener(log, s, SensorManager.SENSOR_DELAY_NORMAL);
  log.init(isXYZ(s));
  panes.closePane();
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/MainActivity.java)

We will discuss the remainder of the `onSensorSelected()` method a bit later in this chapter.

Since `SensorLogFragment` implements `SensorEventListener` — so we can use it with `registerListener()` — we need to implement `onAccuracyChanged()` and `onSensorChanged()`:

```
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
  // unused
}

@Override
public void onSensorChanged(SensorEvent e) {
  Float[] values=new Float[3];

  values[0]=e.values[0];
  values[1]=e.values.length>1 ? e.values[1] : 0.0f;
  values[2]=e.values.length>2 ? e.values[2] : 0.0f;

  adapter.add(values);
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorLogFragment.java)

Once again, we will get into the implementation of `onSensorChanged()` a bit later in this chapter.

The big thing to note now about `onSensorChanged()`, though, is that **the `SensorEvent` object comes from an object pool and gets recycled**. It is not safe for you to hold onto this `SensorEvent` object past the call to `onSensorChanged()`. Hence, you need to do something with the data in the `SensorEvent`, then let go of the `SensorEvent` itself, so that instance can be used again later. This is to help prevent excessive garbage collection, particularly for low-delay requests for sensor readings (e.g., `SENSOR_DELAY_FASTEST`).

Interpreting Sensor Events

The key piece of data in the `SensorEvent` object is `values`. This is a six-element `float` array containing the actual sensor reading. What those values *mean* will vary by sensor. For example:

- For accelerometer readings (e.g., `TYPE_ACCELEROMETER`), the first three elements of the array represent the reported acceleration, in m/s², along the X, Y, and Z axes respectively (X = out the right side of the device, Y = out the top edge of the device, Z = out the screen towards the user's eyes)
- `TYPE_PRESSURE` uses the first element of the `values` array to report the barometric pressure in millibars
- `TYPE_LIGHT` uses the first element of the `values` array to report the light level in lux

And so on.

The `SensorEvent` documentation contains instructions on how to interpret these events on a per-sensor-type basis.

That being said, sensors can be roughly divided into two groups:

1. Sensors whose readings take into account three axes (X/Y/Z). These include `TYPE_ACCELEROMETER`, `TYPE_GRAVITY`, `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION`, and `TYPE_MAGNETIC_FIELD`.
2. Sensors that have simple single-value readings, such as `TYPE_PRESSURE` and `TYPE_LIGHT`

The `isXYZ()` method on `MainActivity` simply returns a `boolean` indicating whether or not this particular `Sensor` is one that uses all three axes (`true`) or not (`false`). As the roster of sensors has changed over the years, it also does some checks based on API level:

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private boolean isXYZ(Sensor s) {
  switch (s.getType()) {
    case Sensor.TYPE_ACCELEROMETER:
    case Sensor.TYPE_GRAVITY:
    case Sensor.TYPE_GYROSCOPE:
    case Sensor.TYPE_LINEAR_ACCELERATION:
    case Sensor.TYPE_MAGNETIC_FIELD:
    case Sensor.TYPE_ROTATION_VECTOR:
      return(true);
  }

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR2) {
    if (s.getType() == Sensor.TYPE_GAME_ROTATION_VECTOR
        || s.getType() == Sensor.TYPE_GYROSCOPE_UNCALIBRATED
        || s.getType() == Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED) {
```

```
        return(true);
      }
    }

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
      if (s.getType() == Sensor.TYPE_GEOMAGNETIC_ROTATION_VECTOR) {
        return(true);
      }
    }

    return(false);
  }
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/MainActivity.java)

Wiring Together the Sample

Overall, this sample app uses the `SlidingPaneLayout` first seen back in the chapter on large-screen support. We have two fragments, in a master-detail pattern, where the "master" will be a list of all available sensors, and the "detail" will be a log of sensor readings from a selected sensor.

Our layout (`res/layout/activity_main.xml`) wires in a `SensorsFragment` (master) and `SensorLogFragment` (detail) in a `SlidingPaneLayout`:

```
<android.support.v4.widget.SlidingPaneLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/panes"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <fragment
    android:id="@+id/sensors"
    android:name="com.commonsware.android.sensor.monitor.SensorsFragment"
    android:layout_width="300sp"
    android:layout_height="match_parent"/>

  <fragment
    android:id="@+id/log"
    android:name="com.commonsware.android.sensor.monitor.SensorLogFragment"
    android:layout_width="400dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

</android.support.v4.widget.SlidingPaneLayout>
```

(from Sensor/Monitor/app/src/main/res/layout/activity_main.xml)

The `SensorsFragment` is reminiscent of `CountriesFragment` from the `SlidingPaneLayout` variant of the `EU4You` sample. The biggest differences are that we use a `SensorListAdapter` for representing the list of sensors, that we use

getSensorList() on our SensorsFragment.Contract class to retrieve the model data, and that we call onSensorSelected() on the contract to report of selections:

```
package com.commonsware.android.sensor.monitor;

import android.hardware.Sensor;
import android.os.Bundle;
import android.view.View;
import android.widget.ListView;
import java.util.List;

public class SensorsFragment extends
    ContractListFragment<SensorsFragment.Contract> {
  static private final String STATE_CHECKED=
      "com.commonsware.android.sensor.monitor.STATE_CHECKED";
  private SensorListAdapter adapter=null;

  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    adapter=new SensorListAdapter(this);
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    setListAdapter(adapter);

    if (state != null) {
      int position=state.getInt(STATE_CHECKED, -1);

      if (position > -1) {
        getListView().setItemChecked(position, true);
        getContract().onSensorSelected(adapter.getItem(position));
      }
    }
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    l.setItemChecked(position, true);

    getContract().onSensorSelected(adapter.getItem(position));
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    state.putInt(STATE_CHECKED, getListView().getCheckedItemPosition());
```

```
  }

  interface Contract {
    void onSensorSelected(Sensor s);

    List<Sensor> getSensorList();
  }
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorsFragment.java)

`SensorListAdapter` illustrates another approach for handling the difference in "activated" row support. The `EU4You` samples used an `activated` style to apply the "activated" support on Android 3.0 and higher. Here, our custom `ArrayAdapter` subclass dynamically chooses between `android.R.layout.simple_list_item_activated_1` (an activated-capable built-in row layout) and the classic `android.R.layout.simple_list_item_1` based upon API level:

```
package com.commonsware.android.sensor.monitor;

import android.hardware.Sensor;
import android.os.Build;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;

class SensorListAdapter extends ArrayAdapter<Sensor> {
  SensorListAdapter(SensorsFragment sensorsFragment) {
    super(sensorsFragment.getActivity(), getRowResourceId(),
          sensorsFragment.getContract().getSensorList());
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View result=super.getView(position, convertView, parent);

    ((TextView)result).setText(getItem(position).getName());

    return(result);
  }

  private static int getRowResourceId() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
      return(android.R.layout.simple_list_item_activated_1);
    }
```

```
    return(android.R.layout.simple_list_item_1);
  }
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorListAdapter.java)

We also have to override `getView()`, as our model is `Sensor`, whose `toString()` is not what we want, so we have to manually populate the list row with `getName()` instead.

`SensorLogFragment` is another `ListFragment`. In particular, though, we set it up for `TRANSCRIPT_MODE_NORMAL`, which means that Android will automatically scroll the `ListView` to the bottom if we add new rows to the list *and* the user has not scrolled up in the list to view past data:

```
  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    getListView().setTranscriptMode(ListView.TRANSCRIPT_MODE_NORMAL);
  }
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorLogFragment.java)

However, we do not initialize our `ListAdapter` in `onActivityCreated()`, as we might normally do. Instead, we have a dedicated `init()` method, to be called by `MainActivity`, where we set up the `SensorLogAdapter` and keep track of whether the `Sensor` that we are logging is designed to report three-dimensional values (`isXYZ` is `true`) or not:

```
  void init(boolean isXYZ) {
    this.isXYZ=isXYZ;
    adapter=new SensorLogAdapter(this);
    setListAdapter(adapter);
  }
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorLogFragment.java)

The `init()` method, in turn, was called by `onSensorSelected()` of `MainActivity`. Hence, whenever the user taps on a sensor, we set up a fresh log. `init()` can do this because `MainActivity` retrieved our `SensorLogFragment` up in `onCreate()`, stashing it in a `log` data member:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
    log=
        (SensorLogFragment)getSupportFragmentManager().findFragmentById(R.id.log);

    panes=findViewById(R.id.panes);
    panes.openPane();
  }
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/MainActivity.java)

Our `onSensorChanged()` method in `SensorLogFragment` copies the `values` from the `SensorEvent` into a separate `Float` array that is our list's model data:

```
  @Override
  public void onSensorChanged(SensorEvent e) {
    Float[] values=new Float[3];

    values[0]=e.values[0];
    values[1]=e.values.length>1 ? e.values[1] : 0.0f;
    values[2]=e.values.length>2 ? e.values[2] : 0.0f;

    adapter.add(values);
  }
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorLogFragment.java)

Most of the sensors will have three readings in the `values` array, but not all will. We are guaranteed at least one element in `values`. So, we normalize our `Float` array to use up to three values from `values`, substituting in `0.0f` for missing elements.

`SensorLogAdapter` uses the `isXYZ` value to determine how it should format the rows:

- For single-value sensors, we just show the first `Float` from the array
- For three-dimensional sensors, we show all three dimensions, plus the "net" (square root of the sum of the squares), separated by slashes

```
  private class SensorLogAdapter extends ArrayAdapter<Float[]> {
    SensorLogAdapter(SensorLogFragment sensorLogFragment) {
      super(sensorLogFragment.getActivity(),
            android.R.layout.simple_list_item_1,
            new ArrayList<Float[]>());
    }

    @SuppressLint("DefaultLocale")
    @Override
```

```
  public View getView(int position, View convertView, ViewGroup parent) {
    TextView row=
        (TextView)super.getView(position, convertView, parent);
    String content=null;
    Float[] values=getItem(position);

    if (isXYZ) {
      content=
          String.format("%7.3f / %7.3f / %7.3f / %7.3f",
                        values[0],
                        values[1],
                        values[2],
                        Math.sqrt(values[0] * values[0] + values[1]
                            * values[1] + values[2] * values[2]));
    }
    else {
      content=String.format("%7.3f", values[0]);
    }

    row.setText(content);

    return(row);
  }
}
```

(from Sensor/Monitor/app/src/main/java/com/commonsware/android/sensor/monitor/SensorLogFragment.java)

The rest of `MainActivity` simply manages the `SlidingPaneLayout`, much like the `EU4YouSlidingPane` sample did.

The Results

When the user taps on a sensor in the list, we get a log of readings:

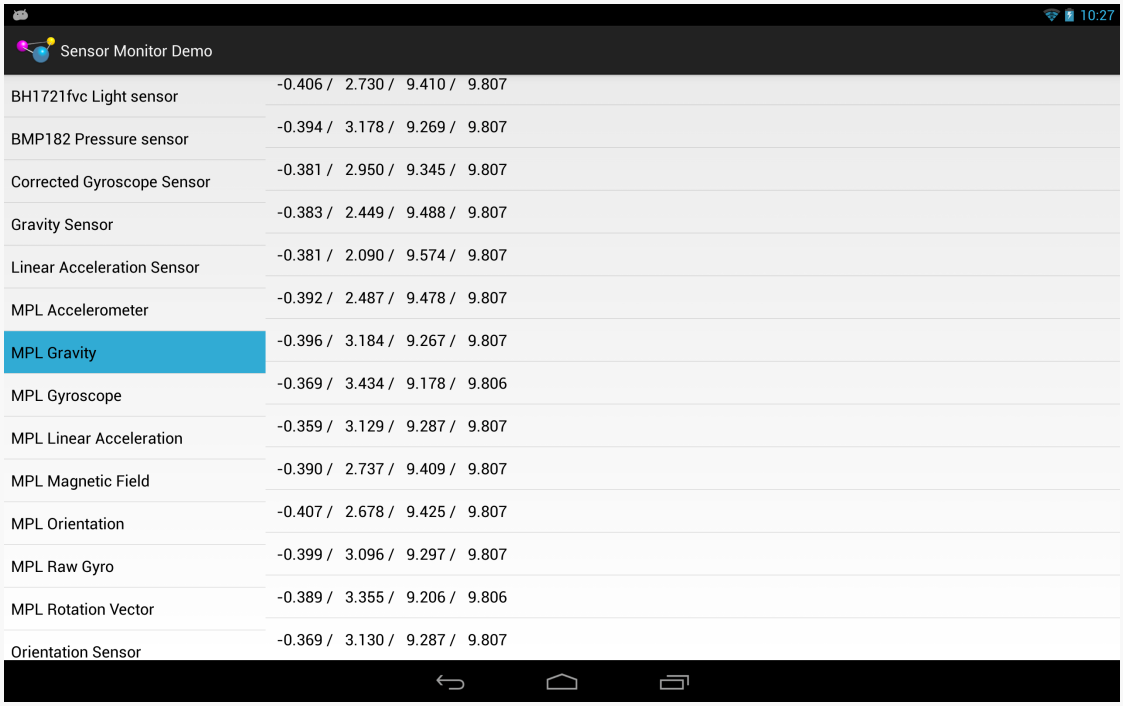


Figure 879: SensorMonitor, On a Nexus 10, Showing Gravity Readings While Being Wiggled by the Author

Batching Sensor Readings

API Level 19 (Android 4.4) added a new feature to the sensor subsystem: batched sensor events. Now, `registerListener()` can take a batch period in microseconds, and Android may elect to deliver events to you delayed by up to that amount of time. The objective will be to reduce the power draw of the sensors, for sensor hardware that supports this sort of batching behavior. Not all hardware will, in which case your requested batch latency will be ignored.

Printing and Document Generation

Mobile devices are continuing to close the gap on capabilities that had formerly been the sole province of desktop systems or servers. After all, if the vision is that people should be able to use phones and tablets *instead of* desktops and notebooks, phones and tablets need to do whatever it is that those people need to have done.

One such capability is the ability to print to networked printers. While various third-party printing options had been available for some time, it is only starting with the Android 4.4 release that the OS and framework itself has support for printing. Hence, at this time, a significant majority of Android devices will be natively capable of printing, and so users will be more likely to expect that your app supports such printing.

As it turns out, the print engine in Android is centered upon the PDF document format, and Android supports converting HTML into PDF, albeit on a somewhat limited basis.

The API seems simple and clean. It actually *is* simple and clean... so long as you are printing very simple contents (bitmaps or HTML). Once you get into anything more complicated than that, the threading alone starts to make things rather messy.

This chapter describes how to use the Android 4.4 print system, including how to print HTML and PDF files. It will also cover how to generate HTML and PDF files, whether for printing or for other purposes (e.g., reports to be emailed or uploaded somewhere).

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Also, you should read the chapter on advanced uses of `WebView`.

The Android Print System

Writing programs that print on desktop operating systems historically has been tedious. The fine-grained control that is needed for high-quality output makes the APIs complicated, and these tend to be only partially masked by high-level wrappers to simplify common scenarios.

Android's print system is no different.

Starting with Android 4.4, you can request access to a `PrintManager` system service (via `getSystemService()`, called on any `Context`). It offers a `print()` method that lets you describe what should be printed, in the form of a `PrintAttributes` (e.g., what size paper are you looking for?) and a `PrintDocumentAdapter`. The latter is responsible for working with Android to actually create the content to be printed.

`print()` returns a `PrintJob`, which you can use to examine the status of the print request. `PrintManager` also offers a `getPrintJobs()` method that returns all of *your* outstanding print requests. Note that you cannot access print jobs from other applications.

Hence, the real complexity of printing lies in the `PrintDocumentAdapter` implementation. This class is responsible for generating a PDF that represents the content to be printed. This leads to four basic ways of working with a `PrintDocumentAdapter`:

1. Have one created for you, such as via a `WebView` for printing HTML content
2. Create one that takes a PDF generated elsewhere and uses it for the output
3. Create one that uses Android's `Canvas`-based PDF generation class, called `PrintedPdfDocument`
4. Use APIs that avoid all of this entirely, such as `printBitmap()` on `PrintHelper`

About the Sample App

The Printing/PrintManager sample project demonstrates all but the `Canvas` option.

The UI is just a large `EditText`, designed for you to type in a message.

The action bar overflow contains four options:

- "Bitmap", to print an image from your device or emulator
- "Web Page", to print the Web page for this book
- "TPS Report", which prints a report containing the message from the `EditText`
- "PDF", which prints a copy of the cover of Version 5.8 of this book, which is packaged in the app as an asset

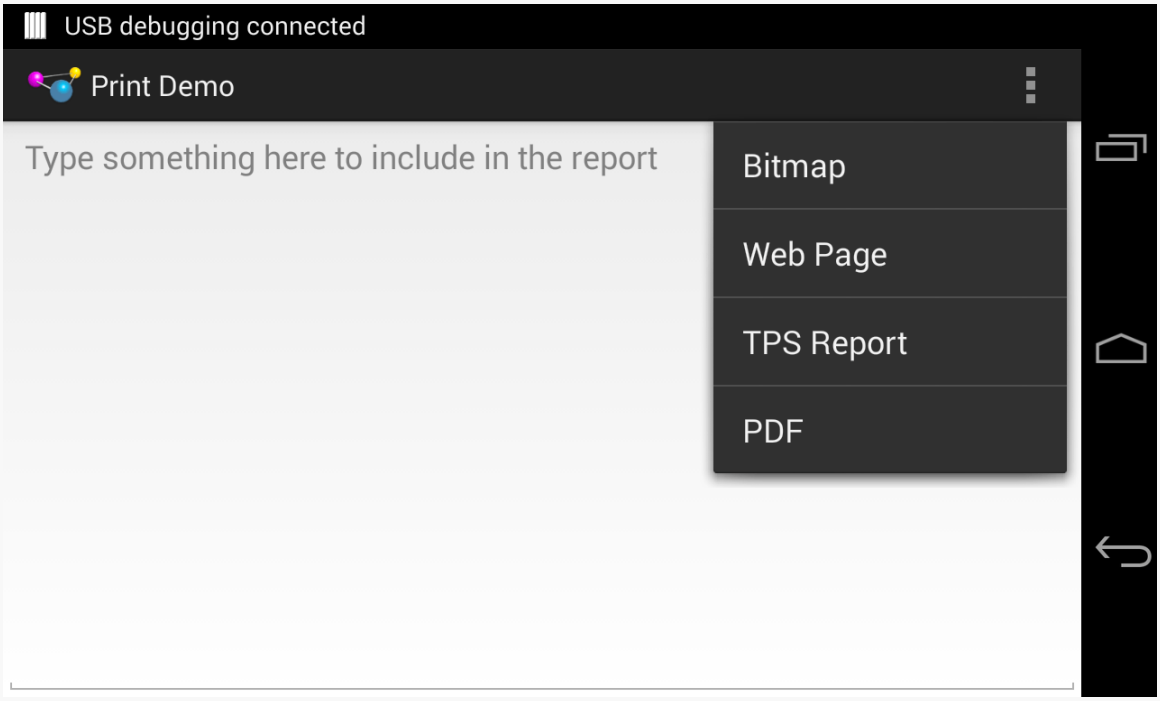


Figure 880: Print Demo App, Showing Overflow

Printing a Bitmap

Google helpfully supplies a `PrintHelper` class in the Android Support package that makes it trivially easy to print a bitmap. Just call `printBitmap()` on the `PrintHelper`, after some minor configuration, and it takes over from there.

In `onOptionsItemSelected()` of the sample app's `MainActivity`, when the user chooses the "Bitmap" item, we call `startActivityForResult()` on an `ACTION_GET_CONTENT` `Intent`, to allow the user to pick an image from the device or emulator:

```
    case R.id.bitmap:
      Intent i=
          new Intent(Intent.ACTION_GET_CONTENT)
            .addCategory(Intent.CATEGORY_OPENABLE)
            .setType("image/*");

      startActivityForResult(i, IMAGE_REQUEST_ID);
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

This, in turn, will trigger a call to `onActivityResult()`, once the user has (presumably) chosen an image:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode == IMAGE_REQUEST_ID
      && resultCode == Activity.RESULT_OK) {
    try {
      PrintHelper help=new PrintHelper(this);

      help.setScaleMode(PrintHelper.SCALE_MODE_FIT);
      help.printBitmap("Photo!", data.getData());
    }
    catch (FileNotFoundException e) {
      Log.e(getClass().getSimpleName(), "Exception printing bitmap",
            e);
    }
  }
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

If the user did indeed choose an image, we create an instance of `PrintHelper`, call `setScaleMode()` to tell it fit the image to the page, and then call `printBitmap()` to print the image.

`setScaleMode()` takes one of two values:

1. `SCALE_MODE_FIT` will show the entire image, blown up as big as possible
2. `SCALE_MODE_FILL` will fill the entire page, at the cost of cropping the image along one axis, if the image's aspect ratio does not match the paper's aspect ratio

`printBitmap()` takes the name of the print job (so the user, when reviewing the outstanding print jobs, knows what it is) and either a `Uri` or a `Bitmap` for the image itself. In the case of a `Uri`, the `Uri` could be malformed, in which case the `FileNotFoundException` may be thrown, which is why we catch it.

What the user sees, after choosing an image to print (and a printer, if the user has more than one available), is a print configuration dialog appear, much like those you might see in a desktop OS:

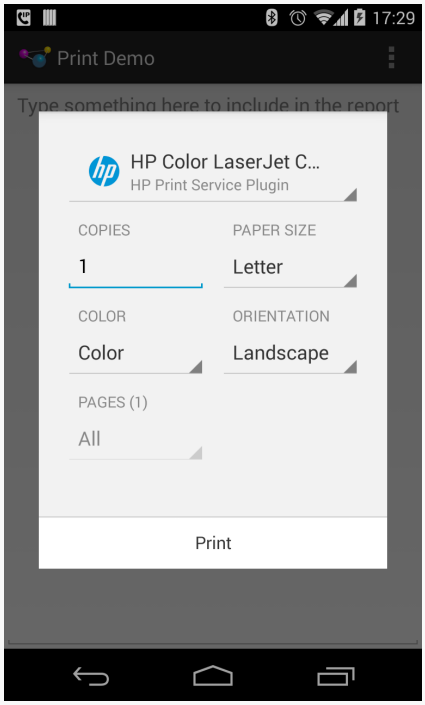


Figure 881: HP Print Configuration Dialog

The dialog itself is provided by Android; the contents of the dialog is provided by a `PrintService` that is responsible for taking our print job and actually dispatching it to the printer.

Here, the user can make typical changes, like portrait/landscape printing and the number of copies, before pressing the "Print" button. At that point, the user's chosen image will be printed.

Note that, in Android 4.4, the print dialog does not work especially well in landscape on smaller screen sizes, forcing the user to scroll to get to all of the widgets, including the "Print" button.

Printing an HTML Document

Printing a bitmap is nice. It is not especially useful, as it implies that we have a bitmap worth printing by itself. That is certainly possible, but it is unlikely. Even in the case where we want to print a photo, there is a very good chance that we will need to print some additional information along with the photo (caption, date when photo was taken, etc.).

Being able to print something over which we have greater control of the rendering would be more useful. The easiest way to do that is to print some HTML. Later in this chapter we will cover how to generate some dynamic HTML representing what you want to print. For the moment, though, let's focus on the printing itself.

Printing and WebView

Starting in API Level 19, `WebView` is capable of participating in the print process. You can load up a `WebView` with your desired content, then print that content.

Some apps will already be using a `WebView` as part of the UI, and that `WebView` will contain what needs to be printed. For example, a Web browser can easily add a "Print" action bar overflow item that would print the contents of the active `WebView` in the browser.

For cases where you want to print something, but you are not using the `WebView` for anything but printing, you do *not* need to add the `WebView` to the UI. You can create a `WebView` instance via its constructor, passing in your `Activity` as the `Context` required by that constructor. You can then populate that `WebView` with what needs to be printed, then print it. That is the technique that the sample application

demonstrates, in part because it is likely to be the more common scenario — only so many apps use a `WebView` in the UI, and more are likely to need to print.

Printing a URL

The sample app's "Web Page" action bar overflow item is tied to an `R.id.web` `MenuItem`. When that is tapped by the user, `onOptionsItemSelected()` calls `printWebPage()` to print a Web page loaded from a URL:

```
private void printWebPage() {
  WebView print=prepPrintWebView(getString(R.string.web_page));

  print.loadUrl("https://commonsware.com/Android");
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

Here, `getString(R.string.web_page)` is returning a string resource that will be used for the name of a print job. `prepPrintWebView()` returns the `WebView` that will be used for printing. `loadUrl()` is the standard `WebView` method for populating the `WebView` from a URL. Note that this causes the sample app to need the `INTERNET` permission, since we are downloading a Web page and its related assets (CSS, images) from the Internet.

You will notice that we are not actually printing anything directly in `printWebPage()`, which may seem a bit odd given the name of the method. That is because we cannot print anything until the page is loaded — after all, it is only then that we have what we want to print.

The job of `prepPrintWebView()` is to arrange to get control when the page is loaded and actually print the desired page:

```
private WebView prepPrintWebView(final String name) {
  WebView result=getWebView();

  result.setWebViewClient(new WebViewClient() {
    @Override
    public void onPageFinished(WebView view, String url) {
      print(name, view.createPrintDocumentAdapter(),
            new PrintAttributes.Builder().build());
    }
  });
```

```
    return(result);
  }
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

`getWebView()` is just a lazy-initialization method, populating a `wv` data member of the activity with a `WebView`. This way, we avoid creating the `WebView` up front, as if the user does not elect to print any HTML, we do not need the `WebView`, and a `WebView` is expensive to initialize:

```
  private WebView getWebView() {
    if (wv == null) {
      wv=new WebView(this);
    }

    return(wv);
  }
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

We are holding onto the `WebView` in a data member to ensure that it will not be garbage-collected. A `WebView` that is part of our UI is being strongly held by its parent in the `View` hierarchy, so we do not normally need to worry about this. However, in this case, we are creating a `WebView` dynamically and are *not* adding it to the UI, so we are responsible for holding onto it, at least as long as is needed. In this sample, we just hold onto it for the rest of the life of the activity.

Back in `prepPrintWebView()`, we call `setWebViewClient()`, to attach an anonymous inner class extending `WebViewClient` to the `WebView`. Back in the chapter introducing `WebView`, we saw `WebViewClient` in the context of `shouldOverrideUrlLoading()`. Another popular method to override on a `WebViewClient` is `onPageFinished()`. This is called when the HTML and related assets (CSS, images, etc.) have been loaded and rendered within the `WebView`. At this point, for the particular URL we are loading, it is safe to print the page.

In `onPageFinished()`, we call a `print()` method on `MainActivity` itself:

```
  private PrintJob print(String name, PrintDocumentAdapter adapter,
                         PrintAttributes attrs) {
    startService(new Intent(this, PrintJobMonitorService.class));

    return(mgr.print(name, adapter, attrs));
  }
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

The first line of `print()` calls `startService()` to start a `PrintJobMonitorService`. We will see more about why we are doing that later in this chapter. For the moment, take it on faith that this service will help ensure that our process stays around long enough for our print job to finish.

The second line of `print()` calls a `print()` method on a `mgr` data member. Here, `mgr` is a `PrintManager`, initialized up in `onCreate()` of the activity, by calling `getSystemService()`, asking for the `PRINT_SERVICE`, and casting the result to be a `PrintManager`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  prose=(EditText)findViewById(R.id.prose);
  mgr=(PrintManager)getSystemService(PRINT_SERVICE);
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

The `print()` method tells the `PrintManager` to go print something. `print()` takes three parameters:

1. The name of the print job, which is kept along with the print job itself in case something (e.g., the print driver) wishes to show the user a roster of print jobs. In our case, this is that string resource passed in as the `name` parameter to `prepPrintWebView()`. That parameter is marked `final`, so the call to `setWebViewClient()` will include the value of that parameter in the anonymous inner class' implementation of `onPageFinished()`.
2. A `PrintDocumentAdapter`. For the case of printing HTML, we get one of those by calling `createPrintDocumentAdapter()` on our populated `WebView`.
3. A `PrintAttributes` object, describing any particular requirements that you have for the printed output (e.g., media size, margins, color/monochrome). If you will let the user control all of that via the print dialog, an empty `PrintAttributes` is fine to use with `print()`. You typically create a `PrintAttributes` by creating a `PrintAttributes.Builder`, calling setters on the `Builder` to configure the `PrintAttributes`, and getting the resulting `PrintAttributes` via a call to `build()`.

And that's it. Android — in particular, `WebView` and its `PrintDocumentAdapter` – takes over from here and prints the Web page.

Limitations and Concerns

Alas, we do not have infinite flexibility with printing HTML from a `WebView`. Here are some limitations and potential problem areas that you will encounter:

- While you can use JavaScript in the loaded HTML, it cannot trigger the print itself using any standard DOM methods.
- Also, if your JavaScript is going to fire off some asynchronous operations, like an AJAX request, bear in mind that `onPageFinished()` does *not* take those operations into account. You will need to use `addJavascriptInterface()` to inject a Java object into the JavaScript realm, then have your asynchronous work arrange to call some method on that Java object, to signal to you that the document is ready for printing.
- Print CSS rules, like headers, footers, page numbers, landscape properties, and the like are ignored at present.
- A `WebView` can only do one print job at a time. Printing occurs asynchronously, and so you have to be careful that you do not accidentally start off a second print job while an earlier one is in process. The `print()` method returns a `PrintJob` that you can use to monitor the print job status, and this object will be covered in a bit more detail later in this chapter. You may wish to set up a `WebView` pool, where you reuse an existing `WebView` only if its associated `PrintJob` is completed, creating a new `WebView` instance if there is no available `WebView` at the moment. Or, you might disable printing options in the UI until the `PrintJob` is done, so you can reuse the `WebView`. The sample app does none of this, to keep things simpler.
- Printing HTML is "an all-or-nothing affair". You cannot print a subset of the HTML, whether denoted by HTML constructs (e.g., `<div>` IDs) or by page numbers. Hence, you need to load into the `WebView` *exactly* what you want to print, no more, no less.

Also, any direct use of `PrintManager` will only work on API Level 19. You will need to ensure that you only try using it on API Level 19+ devices, using Java version guard blocks. You will also need to set your build target (i.e., `compileSdkVersion` in Android Studio) to at least API Level 19 to be able to reference the `PrintManager` and related classes.

Finally, while loading and printing HTML are both intrinsically asynchronous, *generating* HTML locally is not. We will discuss this issue a bit more later in this chapter.

Printing a PDF File

As will be seen in the next section, even if we "hand-roll" our printed output using a `Canvas`, the result seems to be a PDF file. Hence, you would think that the printing framework would provide convenience code to print a PDF file that we obtained by other means.

Alas, that is not the case.

The sample app contains some code demonstrating how this is possible, inspired by this Stack Overflow answer, though it may cut a few corners that Google would prefer not be cut. However, it also illustrates how to create your own `PrintDocumentHandler`, which you will need for any print job not involving a bitmap or HTML.

The PrintDocumentAdapter Protocol

We supply a `PrintDocumentAdapter` to the `print()` method on `PrintManager`. In the HTML case, we got a `PrintDocumentAdapter` from the `WebView`, and so it is Google's job to implement that adapter. Similarly, `PrintHelper` has its own internal implementation of a `PrintDocumentAdapter` that it uses for printing the bitmap.

For anything else, you need to create your own `PrintDocumentAdapter`, or find a third-party implementation that you can perhaps reuse.

`PrintDocumentAdapter`'s job is to supply the `PrintManager` with the content to be printed, in the form of a PDF file. To do that, there are four callback methods that `PrintManager` (and related classes) will call on the `PrintDocumentAdapter`:

1. `onStart()` is called first. If you are planning on using the same `PrintDocumentAdapter` instance for multiple print jobs, this would be a spot to initialize the work for a new job. Otherwise, if you were only planning on using a `PrintDocumentAdapter` instance once, you may as well just put your initialization logic in the constructor.
2. `onLayout()` is called next. Here is where you do enough work to determine what the resulting output will be later on as printing continues. In particular, if you want to provide an accurate page count, this is where you will need to perform the necessary calculations to determine that.
3. `onWrite()` is called next, asking you to write one or more PDF pages out to a supplied `ParcelFileDescriptor` (on which you can create an `OutputStream`).

4. `onFinish()` is called last, when the printing request is completed, so you can free up any necessary resources.

Introducing ThreadedPrintDocumentAdapter

All four of those callback methods are called on the main application thread. Your `onStart()` and `onFinish()` methods need to be fast enough to complete their work on that thread, and that may not be a problem. The work that `onLayout()` and `onWrite()` do may take a while, though, and so the protocol is designed to allow you to do that work on a background thread. Both methods are passed a callback object that you use to pass along the results of your work, and both are passed a `CancellationSignal` to indicate if the user cancels the print job while you are doing the work.

What `PrintDocumentAdapter` does *not* do is actually give you a thread to use.

So, the sample app contains a `ThreadedPrintDocumentAdapter` that moves the `onLayout()` and `onFinish()` work to a background thread:

```
package com.commonsware.android.print;

import android.content.Context;
import android.os.Bundle;
import android.os.CancellationSignal;
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

abstract class ThreadedPrintDocumentAdapter extends
    PrintDocumentAdapter {
  abstract LayoutJob buildLayoutJob(PrintAttributes oldAttributes,
                                    PrintAttributes newAttributes,
                                    CancellationSignal cancellationSignal,
                                    LayoutResultCallback callback,
                                    Bundle extras);

  abstract WriteJob buildWriteJob(PageRange[] pages,
                                  ParcelFileDescriptor destination,
                                  CancellationSignal cancellationSignal,
                                  WriteResultCallback callback,
                                  Context ctxt);
```

```
private Context ctxt=null;
private ExecutorService threadPool=Executors.newFixedThreadPool(1);

ThreadedPrintDocumentAdapter(Context ctxt) {
  this.ctxt=ctxt;
}

@Override
public void onLayout(PrintAttributes oldAttributes,
                     PrintAttributes newAttributes,
                     CancellationSignal cancellationSignal,
                     LayoutResultCallback callback, Bundle extras) {
  threadPool.submit(buildLayoutJob(oldAttributes, newAttributes,
                                   cancellationSignal, callback,
                                   extras));
}

@Override
public void onWrite(PageRange[] pages,
                    ParcelFileDescriptor destination,
                    CancellationSignal cancellationSignal,
                    WriteResultCallback callback) {
  threadPool.submit(buildWriteJob(pages, destination,
                                  cancellationSignal, callback, ctxt));
}

@Override
public void onFinish() {
  threadPool.shutdown();

  super.onFinish();
}

protected abstract static class LayoutJob implements Runnable {
  PrintAttributes oldAttributes;
  PrintAttributes newAttributes;
  CancellationSignal cancellationSignal;
  LayoutResultCallback callback;
  Bundle extras;

  LayoutJob(PrintAttributes oldAttributes,
            PrintAttributes newAttributes,
            CancellationSignal cancellationSignal,
            LayoutResultCallback callback, Bundle extras) {
    this.oldAttributes=oldAttributes;
    this.newAttributes=newAttributes;
    this.cancellationSignal=cancellationSignal;
```

```
      this.callback=callback;
      this.extras=extras;
    }
  }

  protected abstract static class WriteJob implements Runnable {
    PageRange[] pages;
    ParcelFileDescriptor destination;
    CancellationSignal cancellationSignal;
    WriteResultCallback callback;
    Context ctxt;

    WriteJob(PageRange[] pages, ParcelFileDescriptor destination,
             CancellationSignal cancellationSignal,
             WriteResultCallback callback, Context ctxt) {
      this.pages=pages;
      this.destination=destination;
      this.cancellationSignal=cancellationSignal;
      this.callback=callback;
      this.ctxt=ctxt;
    }
  }
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/ThreadedPrintDocumentAdapter.java)

This class uses a single-thread thread pool, managed by an `ExecutorService`. In principle, a well-written `PrintDocumentAdapter` could handle multiple print jobs in parallel — if you attempt this and are using `ThreadedPrintDocumentAdapter` for inspiration, simply increase the size of the thread pool.

The `onLayout()` and `onWrite()` methods package up their parameters (described in the next section) into job objects. Those objects implement `Runnable`, and they are then handed to the `ExecutorService` to be run on the next-available thread. `onFinish()` shuts down the `ExecutorService`, though if you wanted to use the `ThreadedPrintDocumentAdapter` for multiple print jobs, you would come up with some other logic to clean up the `ExecutorService` when you were done with *all* of the jobs.

Subclasses of `ThreadedPrintDocumentAdapter` need to:

- Create subclasses of the `LayoutJob` and `WriteJob` static inner classes, implementing their respective `run()` methods, to do the work required of `onLayout()` and `onWrite()`

- Implement `buildLayoutJob()` and `buildWriteJob()` methods that return instances of those custom subclasses

(fans of dependency injection no doubt can find better solutions for wiring up a `ThreadedPrintDocumentAdapter`)

A PdfDocumentAdapter

However, we still need to actually be able to print a PDF, which `ThreadedPrintDocumentAdapter` does not do on its own. The sample app also has a `PdfDocumentAdapter`, which extends `ThreadedPrintDocumentAdapter` and demonstrates a crude way of printing a PDF through the `PrintDocumentAdapter` protocol.

`PdfDocumentAdapter` does not use `onStart()` or `onFinish()`. And, since the `onLayout()` and `onWrite()` methods are handled by `ThreadedPrintDocumentAdapter`, `PdfDocumentAdapter` does not have those either.

It does, however, have the `buildLayoutJob()` and `buildWriteJob()` methods required by `ThreadedPrintDocumentAdapter`. These return instances of a `PdfLayoutJob` and `PdfWriteJob`, respectively:

```
@Override
LayoutJob buildLayoutJob(PrintAttributes oldAttributes,
                         PrintAttributes newAttributes,
                         CancellationSignal cancellationSignal,
                         LayoutResultCallback callback, Bundle extras) {
  return(new PdfLayoutJob(oldAttributes, newAttributes,
                          cancellationSignal, callback, extras));
}

@Override
WriteJob buildWriteJob(PageRange[] pages,
                       ParcelFileDescriptor destination,
                       CancellationSignal cancellationSignal,
                       WriteResultCallback callback, Context ctxt) {
  return(new PdfWriteJob(pages, destination, cancellationSignal,
                         callback, ctxt));
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/PdfDocumentAdapter.java)

`PdfLayoutJob` needs to fulfill the bulk of the `onLayout()` contract:

- Monitor the `CancellationSignal` and call `onLayoutCancelled()` on the supplied `LayoutResultCallback` if the job has been canceled
- Populate a `PrintDocumentInfo` object to provide metadata about the document to be printed, and pass that to `onLayoutFinished()` on the `LayoutResultCallback`

```
private static class PdfLayoutJob extends LayoutJob {
  PdfLayoutJob(PrintAttributes oldAttributes,
               PrintAttributes newAttributes,
               CancellationSignal cancellationSignal,
               LayoutResultCallback callback, Bundle extras) {
    super(oldAttributes, newAttributes, cancellationSignal, callback,
          extras);
  }

  @Override
  public void run() {
    if (cancellationSignal.isCanceled()) {
      callback.onLayoutCancelled();
    }
    else {
      PrintDocumentInfo.Builder builder=
          new PrintDocumentInfo.Builder("CHANGE ME PLEASE");

      builder.setContentType(PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
             .setPageCount(PrintDocumentInfo.PAGE_COUNT_UNKNOWN)
             .build();

      callback.onLayoutFinished(builder.build(),
                                !newAttributes.equals(oldAttributes));
    }
  }
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/PdfDocumentAdapter.java)

`PdfLayoutJob` also has access to two `PrintAttributes` objects, the "old" attributes and the "new" attributes. In principle, `onLayout()` could be called a couple of times, perhaps based upon changes the user makes in the print dialog. These `PrintAttributes` objects describe the nature of the output, including things like page size and margins. `PdfLayoutJob` totally ignores these, because the PDF is a packaged asset in this case and cannot be changed. If you are dynamically generating a PDF file, you may wish to pay attention to the new `PrintAttributes` and take them into account.

`PdfLayoutJob` also has access to a `Bundle` of "extras", not unlike the "extras" associated with an `Intent`. At the present time, there is only one semi-documented "extra", `EXTRA_PRINT_PREVIEW`, which will be `true` if `onLayout()` is being called to generate a print preview of the printed output, `false` otherwise.

What `PdfLayoutJob` does do is create a `PrintDocumentInfo.Builder` to set up a `PrintDocumentInfo` object indicating that:

- The output is a "document" (`CONTENT_TYPE_DOCUMENT`) versus a "photo" (`CONTENT_TYPE_PHOTO`) or "unknown" (`CONTENT_TYPE_UNKNOWN`). This information is passed to the `PrintService` that functions as a bridge between `PrintManager` and the printer, and the `PrintService` might optimize output based upon this setting (e.g., lower quality print output for a "document" instead of a "photo").
- The page count of the output is unknown (`PAGE_COUNT_UNKNOWN`). In principle, the page count is known, insofar as the PDF that will be printed is an asset baked into the app, and so we could hard-code the page count in addition to hard-coding other details (like the asset's filename).

The boolean second parameter to `onLayoutFinished()` is supposed to be `true` if the layout changed, `false` otherwise. In practice, the value does not seem to matter on the first `onLayout()` call. The implementation here compares the two `PrintAttributes` objects using `equals()`.

The last piece is the `PdfWriteJob`, which performs the work required of the `onWrite()` callback:

```
private static class PdfWriteJob extends WriteJob {
  PdfWriteJob(PageRange[] pages, ParcelFileDescriptor destination,
              CancellationSignal cancellationSignal,
              WriteResultCallback callback, Context ctxt) {
    super(pages, destination, cancellationSignal, callback, ctxt);
  }

  @Override
  public void run() {
    InputStream in=null;
    OutputStream out=null;

    try {
      in=ctxt.getAssets().open("cover.pdf");
      out=new FileOutputStream(destination.getFileDescriptor());
```

```
        byte[] buf=new byte[16384];
        int size;

        while ((size=in.read(buf)) >= 0
            && !cancellationSignal.isCanceled()) {
          out.write(buf, 0, size);
        }

        if (cancellationSignal.isCanceled()) {
          callback.onWriteCancelled();
        }
        else {
          callback.onWriteFinished(new PageRange[] { PageRange.ALL_PAGES });
        }
      }
      catch (Exception e) {
        callback.onWriteFailed(e.getMessage());
        Log.e(getClass().getSimpleName(), "Exception printing PDF", e);
      }
      finally {
        try {
          in.close();
          out.close();
        }
        catch (IOException e) {
          Log.e(getClass().getSimpleName(),
                "Exception cleaning up from printing PDF", e);
        }
      }
    }
  }
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/PdfDocumentAdapter.java)

At its core, `PdfWriteJob` simply writes our PDF (culled from a `cover.pdf` asset) to an `OutputStream`. The `OutputStream` is built from the `ParcelFileDescriptor`, indicating where the PDF content should be written to.

The `InputStream`-to-`OutputStream` "bucket brigade" is augmented with checks on the `CancellationSignal`, to abandon the loop if the print job was canceled by the user. At the end, we call one of three methods on the `WriteResultCallback`:

- `onWriteCancelled()` if the `CancellationSignal` indicates that the job was canceled
- `onWriteFinished()` if everything succeeded

- `onWriteFailed()` (with an error message) if there was some problem, such as failed I/O

`PdfWriteJob` has access to a `PageRange` array, representing the particular pages out of a larger document to be printed. The parameter to `onWriteFinished()` is another `PageRange` array that should indicate what pages were printed. Once again, since the PDF is fixed, `PdfWriteJob` ignores the input `PageRange` array, and it indicates that we wrote all pages (`PageRange.ALL_PAGES`) in the output. In principle, if you have more control over your environment, you should only print the requested pages, in which case the output parameter to `onWriteFinished()` might be the same array as was passed into `onWrite()`.

Using PdfDocumentAdapter

Back in `MainActivity`, the "PDF" action bar overflow item triggers a call to `print()` on the `PrintManager`, supplying our `PdfDocumentAdapter` and another empty `PrintAttributes`:

```
      case R.id.pdf:
        print("Test PDF",
              new PdfDocumentAdapter(getApplicationContext()),
              new PrintAttributes.Builder().build());
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

The `PdfDocumentAdapter` needs a `Context`, in order to access the `cover.pdf` asset. If your PDF file is being generated, or is saved as a file on external storage, you would not need this. Since it is theoretically possible that our activity could be destroyed while the printing is going on in background threads, rather that briefly leak an `Activity`, we provide the `Application Context` to `PdfDocumentAdapter`, as that is a singleton and cannot be leaked.

The result of all of this is that when the user chooses the "PDF" action bar overflow item, the book cover copy is printed.

Printing Using a Canvas

What Google *really* wants you to do — if bitmaps and HTML are insufficient – is to create PDF documents using `PrintedPdfDocument` and a `Canvas`.

The concept is simple:

- Create a `PrintedPdfDocument` instance, given a `PrintAttributes` that describes the page size, margins, etc.
- Call `startPage()` to add a page to the document, which returns a `PdfDocument.Page`
- Call `getCanvas()` on the `Page` and use the standard Android 2D drawing APIs to draw lines, text, shaded areas, and so forth
- Call `finishPage()` on the `PdfPrintedDocument` when you are done rendering that page
- Repeat the preceding three steps for all needed pages
- Call `writeTo()` on the `PrintedPdfDocument` to write the PDF to an `OutputStream`, such as the one you get from the `ParcelFileDescriptor` in the `onWrite()` callback of your `PrintDocumentAdapter`
- Call `close()` on the `PrintedPdfDocument` when you are done

For example, let's look at the `onWrite()` implementation used by `PrintHelper` to print a bitmap:

```
@Override
public void onWrite(PageRange[] pageRanges, ParcelFileDescriptor fileDescriptor,
                    CancellationSignal cancellationSignal,
                    WriteResultCallback writeResultCallback) {
    PrintedPdfDocument pdfDocument = new PrintedPdfDocument(mContext,
            mAttributes);
    try {

        Page page = pdfDocument.startPage(1);
        RectF content = new RectF(page.getInfo().getContentRect());

        // Compute and apply scale to fill the page.
        Matrix matrix = getMatrix(mBitmap.getWidth(), mBitmap.getHeight(),
                content, fittingMode);

        // Draw the bitmap.
        page.getCanvas().drawBitmap(mBitmap, matrix, null);

        // Finish the page.
        pdfDocument.finishPage(page);

        try {
            // Write the document.
            pdfDocument.writeTo(new FileOutputStream(
                    fileDescriptor.getFileDescriptor()));
            // Done.
            writeResultCallback.onWriteFinished(
                    new PageRange[]{PageRange.ALL_PAGES});
```

```
        } catch (IOException ioe) {
            // Failed.
            Log.e(LOG_TAG, "Error writing printed content", ioe);
            writeResultCallback.onWriteFailed(null);
        }
    } finally {
        if (pdfDocument != null) {
            pdfDocument.close();
        }
        if (fileDescriptor != null) {
            try {
                fileDescriptor.close();
            } catch (IOException ioe) {
                /* ignore */
            }
        }
    }
}
```

(note: the preceding code snippet is Copyright (C) 2013 The Android Open Source Project)

Here, they:

- Create the `PrintedPdfDocument`
- Add a page using `startPage()`
- Calculate a scaling `Matrix` based upon the image size, the page size, and the scale type (`FIT` or `FILL`)
- Draw the bitmap on the `Canvas` using that `Matrix`
- Finish the page
- Write the result to an `OutputStream` for the supplied `ParcelFileDescriptor`
- Close the document

Curiously, they do not do this work in a background thread, though the `onLayout()` implementation does use a background thread (since the image `Uri` may require an Internet download).

If you are comfortable with the `Canvas` API, writing PDF pages is much the same as drawing to your custom `View`. On the other hand, Android's `Canvas` API is not the same as any other drawing system's API, so there will be distinct differences from any other 2D drawing API that you might have used previously.

Print Jobs

The `print()` method that we have been calling on `PrintManager` returns a `PrintJob`, representing the print job. This object has a number of status inquiry methods, including (in rough order of when the events occur):

- `isStarted()`
- `isQueued()` (i.e., waiting for the print system to process it)
- `isBlocked()` (i.e., permanently stuck, but needs to be canceled)
- `isCompleted()`
- `isFailed()`
- `isCancelled()`

It also has a `cancel()` method that you can call to cancel the print job (e.g., based on user request). `PrintJob` also offers a `restart()` method that you can use to re-try a failed (but not canceled) print job.

What `PrintJob` does not have is a listener interface to be proactively notified when the job changes state.

`PrintManager` also has `getPrintJobs()`, which will return a list of the `PrintJob` objects representing the jobs you have requested in this process, rather than having to keep track of all of those yourself.

Printing, Threads, and Services

If you are going to create a report in HTML, you will want to consider doing that work in an `AsyncTask`'s `doInBackground()` method, so the I/O involved in creating the report happens in the background. However, `PrintManager` requires that `print()` be called on the main application thread, so you would call `print()` from `onPostExecute()` of the `AsyncTask`.

Similarly, if you are creating your own `PrintDocumentAdapter`, you will want to consider moving the `onLayout()` and `onWrite()` work into background threads, such as is illustrated in the sample app via `ThreadedPrintDocumentAdapter`.

The problem with bare threads or an `AsyncTask` is that they do not indicate to Android that your process is still doing some work. It is possible that the user could request that you print something, then switch to another app (e.g., HOME, recent-

tasks list). Android might consider your process to be relatively low priority and could terminate it before your print job completes.

The obvious solution is to involve a service, perhaps even a foreground service, to indicate to Android that your process is doing work that the user will notice if it does not complete. You could start the service when you do the print job, and then stop the service when the print job is completed, to return your process to normal priority.

However, actually having a service do the printing is a serious pain:

- `WebView`'s `PrintDocumentAdapter` really wants the `Context` that created the `WebView` to be an `Activity`
- The key parameters to `onLayout()` and `onWrite()` are not `Parcelable` and so cannot be passed in `Intent` extras via `startService()` to the service

One possibility would be to create a `PrintJobMonitorService`, which is what the sample app does. `PrintJobMonitorService` takes advantage of that `listPrintJobs()` method on `PrintManager` to keep tabs on all of our requested print jobs. So long as there is one or more print jobs in an active state, the service keeps running. Otherwise, the service stops. Hence, while the service is not actually doing the printing, it is running while the printing is going on, flagging to the OS to leave our process alone during this critical juncture.

```
package com.commonsware.android.print;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.SystemClock;
import android.print.PrintJob;
import android.print.PrintJobInfo;
import android.print.PrintManager;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class PrintJobMonitorService extends Service implements Runnable {
  private static final int POLL_PERIOD=3;
  private PrintManager mgr=null;
  private ScheduledExecutorService executor=
      Executors.newSingleThreadScheduledExecutor();
  private long lastPrintJobTime=SystemClock.elapsedRealtime();
```

```
  @Override
  public void onCreate() {
    super.onCreate();

    mgr=(PrintManager)getSystemService(PRINT_SERVICE);
    executor.scheduleAtFixedRate(this, POLL_PERIOD, POLL_PERIOD,
                                 TimeUnit.SECONDS);
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    return(super.onStartCommand(intent, flags, startId));
  }

  @Override
  public void onDestroy() {
    executor.shutdown();

    super.onDestroy();
  }

  @Override
  public void run() {
    for (PrintJob job : mgr.getPrintJobs()) {
      if (job.getInfo().getState() == PrintJobInfo.STATE_CREATED
          || job.isQueued() || job.isStarted()) {
        lastPrintJobTime=SystemClock.elapsedRealtime();
      }
    }

    long delta=SystemClock.elapsedRealtime() - lastPrintJobTime;

    if (delta > POLL_PERIOD * 2) {
      stopSelf();
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/PrintJobMonitorService.java)

`PrintJobMonitorService` uses a single-thread `ScheduledExecutorService`, to get control every three seconds in its `run()` method. The `run()` method iterates over the

`PrintJob` objects associated with our app and looks for any that are in one of three states:

- "started", meaning that printing has begun
- "queued", meaning that the user has accepted the print dialog values, but printing has not yet started
- "created", meaning that the job has been created, but it is not yet considered queued, such as when the print dialog is up on the screen

The first two states have simple test methods on `PrintJob` (`isStarted()` and `isQueued()`). The "created" state does not, for some reason, so we have to get the underlying `PrintJobInfo` object and manually check its state (`getState()`) to see if it is started (`PrintJobInfo.STATE_STARTED`).

`PrintJobMonitorService` tracks the last time we saw an in-progress print job. If we have gone through two three-second polling periods without any in-progress print jobs, the service assumes that it is no longer needed and calls `stopSelf()`.

Printing Prior to Android 4.4

Before Android 4.4, printing in Android was limited and clunky.

The primary approach was to use Google Cloud Print. In effect, Google Cloud Print is a Web-managed print server. You would teach Google how to talk to your printers, and then any authorized device could print to those printers. By sharing your content (particularly PDFs) via `ACTION_SEND`, the user could choose Google Cloud Print as an option if they had Google Cloud Print set up for their device and printer. Note that the Android 4.4 printing framework includes a `PrintService` that works with Google Cloud Print, so users who have set up Google Cloud Print can still use it even with the new printing framework.

Various printer manufacturers or third parties also created their own apps that would fill a similar role, albeit perhaps working with printers on the local network. Or, you could write your own low-level code to talk to a network printer via relevant printing protocols like IPP, though this would be unpleasant at best.

HTML Generation

Earlier in this chapter, we saw how to print HTML. However, the HTML we printed was loaded from a URL. That is fine, but, as with printing bitmaps, it may not be a

very popular scenario. What will be more likely is that you want to print some sort of report, generated on the device. And, since printing using the `Canvas` is a bit complicated, creating the report via HTML may be an easier route to take.

The typical approach for this involves creating an HTML template that sets up the basic page (e.g., references to CSS), then uses some sort of "macros" in the template to indicate portions that should be replaced dynamically with something from outside of the template.

This approach has been used since the early days of the original "dot-com revolution" of the 1990's, pioneered by tools like Cold Fusion. In Java, there are any number of available template engines.

However, for HTML, it is reasonably likely that a Web designer is going to want to get involved, to style the report. Ideally, you choose a template engine that is either something the designer is already using, or is one that is something the designer might wish to use elsewhere in the future. Forcing the designer to learn some new template syntax, just for the purposes of creating these reports, may not be the best use of the designer's time (or your time, for answering all of the designer's questions).

One of the more popular template structures used today use braces (a.k.a., curly brackets) as the macro delimiters (e.g., `{{ something }}`). In particular, the macro syntax popularized by mustache is used by many template engine implementations. There is a very good chance that your Web designer already has used mustache-style templates, or at least has heard about them.

And, conveniently enough, there is a Java implementation – jmustache — that is Android-friendly. The sample app in this chapter implements a "TPS Report" that is generated from a mustache template using jmustache.

Adding jmustache To Your App

The "Get It" section of the jmustache documentation contains up-to-date instructions for adding it to a project.

Developers using the Android Gradle Plugin — including Android Studio users – should reference the Maven Central artifact (`com.samskivert:jmustache`) from `build.gradle`.

Writing the Report Template

A report template for jmustache can be a `String` or a `Reader`, with the latter allowing you to pull in files, assets, or raw resources (the latter two via an `InputStreamReader`).

In the case of the sample app, the template is small, and is packaged as a string resource. However, since the template involves HTML tags, we have to use `CDATA` notation to allow those tags to be left alone within the XML of the string resource:

```
  <string name="report_body">
<![CDATA[
<html>
<body>
<h1>TPS Report for: {{reportDate}}</h1>
<p>Here are the contents of this week\'s TPS report:</p>
<p>{{message}}</p>
<p>If you have any questions regarding this report, please
do <b>not</b> ask Mark Murphy.</p>
</body>
</html>
]]>
```

Figure 882: TPS Report Mustache Template

The template contains `{{reportDate}}` and `{{message}}` variables to be replaced at runtime with dynamic data from our app. Also note that, despite the `CDATA`, we still need to escape the apostrophe with a leading backslash (`\'`).

Creating a Report Context

What will fill in the `{{reportDate}}` and `{{message}}` variables will be values from a "context". Here, "context" is not referring to `Context`, but rather an object that we pass to jmustache to serve as the source of data to blend into the report.

jmustache has fairly flexible rules for how it can resolve template variables, including calling Java getter methods based on the variable names. Hence, we can create a "context" that has `getReportDate()` and `getMessage()` methods, such as the `TpsReportContext` class in the sample app:

```
private static class TpsReportContext {
  private static final SimpleDateFormat fmt=
      new SimpleDateFormat("yyyy-MM-dd", Locale.US);
```

```
  String msg;

  TpsReportContext(String msg) {
    this.msg=msg;
  }

  @SuppressWarnings("unused")
  String getReportDate() {
    return(fmt.format(new Date()));
  }

  @SuppressWarnings("unused")
  String getMessage() {
    return(msg);
  }
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

Printing the Report

The "TPS Report" action bar overflow item eventually routes to a `printReport()` method on `MainActivity`:

```
private void printReport() {
  Template tmpl=
      Mustache.compiler().compile(getString(R.string.report_body));
  WebView print=prepPrintWebView(getString(R.string.tps_report));

  print.loadData(tmpl.execute(new TpsReportContext(prose.getText()
                                                        .toString())),
                 "text/html", "UTF-8");
}
```

(from Printing/PrintManager/app/src/main/java/com/commonsware/android/print/MainActivity.java)

The first statement creates a jmustache `Template` object representing the report template. This is created by getting the singleton `compiler()` from `Mustache`, and calling `compile()` on it to interpret the string resource. Note that since this `Template` only depends upon the string resource, we could cache the `Template`, rebuilding it only on configuration changes, if desired.

Note that we load the template on the main application thread, as `printReport()` is called from `onOptionsItemSelected()`. For a small string resource, that is OK. If you are loading a more complex report template, you will want to do that in a background thread.

The second statement mirrors one from printing the Web page from before, where we call `prepPrintWebView()` to lazy-create our `WebView` and set it up to print when the page is loaded. Here, we use a different print job name than before, one reflecting the fact that this is a TPS report.

Finally, we use `execute()` on the `Template` to generate our HTML for printing, then pass that HTML to the `loadData()` method on `WebView`. `execute()` takes our "context" `Object`, which in this case is an instance of our `TpsReportContext` class, with the value typed into the `EditText` widget in our UI as the "message" to go into the report.

Note that we `execute()` the `Template` on the main application thread as well as having loaded it on that thread in the first place. Once again, the more complex the report, the more likely it is that you will want to move this logic into a background thread. However, remember that `print()` needs to be called on the main application thread.

The result is that the user gets a printed TPS report, containing today's date and whatever message they typed into the `EditText`.

PDF Generation Options

Perhaps you feel that generating HTML does not give you enough control, yet using the `Canvas` options directly was *too* much control. Perhaps you then think that generating a PDF to print, using something other than `PdfDocument`, is the right answer. Or perhaps you are generating a PDF for other reasons, such as to use with `ACTION_SEND` as output from your app.

You have two basic options for getting this PDF: generate it on the device, or offload the generation to a server.

There are various open source and commercial libraries for generating PDF on Android. The best-known open source Java PDF library – iText — has as dedicated Android version (iTextG), though the AGPL license may make it unsuitable for your use case. The commercial libraries range from fixed-price to per-device licenses. How much advantage these have over using `PrintedPdfDocument` from the Android SDK depends upon your needs.

If the bulk of the data needed for generating the PDF resides on a server, rather than downloading that data and using an underpowered Android device to create the

PDF, you could upload the device-specific data to the server, have it create the PDF, and download the result from the server. There are plenty of server-side PDF generation tools, ranging from open source (e.g., wkhtmltopdf, unoconv, prawn) to commercial (e.g., Prince, used to generate the PDF edition of this book). You also get to work in your preferred programming language, in case that is not Java, and perhaps leverage the PDF generation logic for other uses (e.g., generate reports from your Web app).

Basic Bluetooth RFCOMM

For short-range communications, Bluetooth is fairly popular. It is widespread, available on mobile devices, notebooks, and many Internet of Things platforms. It performs reasonably well, at least for moderate amounts of data. Android has a variety of classes in the Android SDK for adding Bluetooth communications to an app.

However, Bluetooth overall is a *vast* topic. The documentation for the Android SDK classes is spotty. And it can be fairly difficult to make sense of how all the different pieces are supposed to plug in together.

In this chapter, we will explore a sample app that demonstrates Bluetooth communications between two Android devices and use that to see how to work with Bluetooth on Android. For extra fun, we will also peek a bit at how things differ when you try to use Bluetooth on an Android Things device, such as a Raspberry Pi.

Prerequisites

This chapter makes use of RxJava, foreground services, `RecyclerView`, and data binding.

If you want to run the sample app, you will need two Android 5.0+ devices, each with working Bluetooth.

A Quick Bit of Scope

As mentioned, Bluetooth is vast, much more than can be covered in a single chapter.

This chapter will focus on Bluetooth, not Bluetooth Low Energy (BLE). When most people think of Bluetooth, they are thinking of "full" Bluetooth. BLE is designed for low-power environments and lower data throughput.

This chapter will focus on RFCOMM. Bluetooth is based around "profiles", which describe particular standards of data exchange between parties. If you think of Bluetooth as being HTTPS, a Bluetooth profile is a particular Web service API. RFCOMM is a general-purpose mechanism designed for communications that fall outside any standard profile.

And, this chapter will focus on one particular recipe for using Bluetooth. As with many of the book examples, the code shown here is not bulletproof, but is here to illustrate the use of various APIs and concepts. A production-grade app will need to handle concerns that lie outside the scope of the chapter, such as:

- What if we try sending significant amounts of data, not just short strings, between the devices?
- What if the devices that had been in communication move out of range of one another?
- What if Bluetooth is turned off on one of the devices, or if that device goes into airplane mode?
- How many simultaneous Bluetooth connections can an app manage, and how do we handle cases where we hit the limit?

About the Sample App

The sample app – `Bluetooth/RxEcho` – is somewhat more complex than are many in this book. The explanation of how the app works may be somewhat clearer if we take a look at the app overall first, before diving in the details.

The User Experience

When you launch the app, Android will prompt you to allow the app to detect the device's location. As we will see later in the chapter, Bluetooth has some interesting permission requirements.

If you agree to the permission, you then may get a dialog asking you if you want to enable Bluetooth, if it is not already enabled. If you decline, the app shuts down, as it cannot really do anything useful.

If you grant the location permission and enable Bluetooth, the app will present a largely empty UI. The fun begins with the action bar overflow menu, where there is a “Server” checkable item, a “Discover” item, and an “Allow Discovery” item.

Designate one of your two test devices as the “client” and the other as the “server”. Both will need the app up and running.

On the server, click the “Server” action bar item. This will spawn a `Notification` letting you know that something is running... a bit loudly:

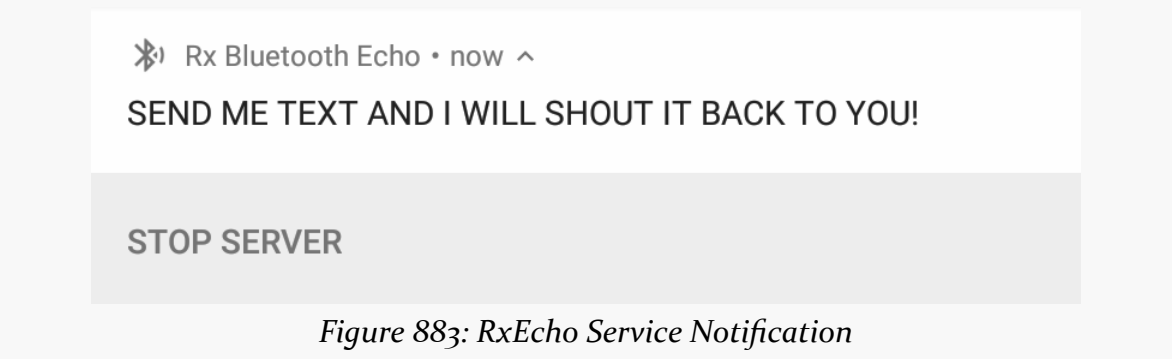


Figure 883: RxEcho Service Notification

Later, when you are done with the sample app, you can stop the server either through the action bar overflow item (unchecking it) or by clicking the “Stop Server” action in the `Notification`.

Then, on the server, from the action bar overflow menu, choose “Allow Discovery”. This will pop up a dialog box asking you if you are willing to allow other devices to discover this one, for 120 seconds. If you accept the dialog... nothing visible happens, but your device will now be discoverable by other Bluetooth devices.

During those 120 seconds, on the client, choose “Discover” from the action bar overflow menu. You should see an entry pop up with the Bluetooth MAC address of the server device:

Figure 884: RxEcho Client, Showing Discovered Device

You have no good way of knowing that this is the Bluetooth MAC address of your other device, but most likely there is nothing else discoverable right now that matches what the app is looking for, and so most likely what you find will be your server device.

Then, tap on that MAC address, to bring up a screen dedicated to that device:

Figure 885: RxEcho Client, Showing the Selected Device

Click the “Connected” switch. After a short delay, you should see a pairing dialog appear on *both* devices:

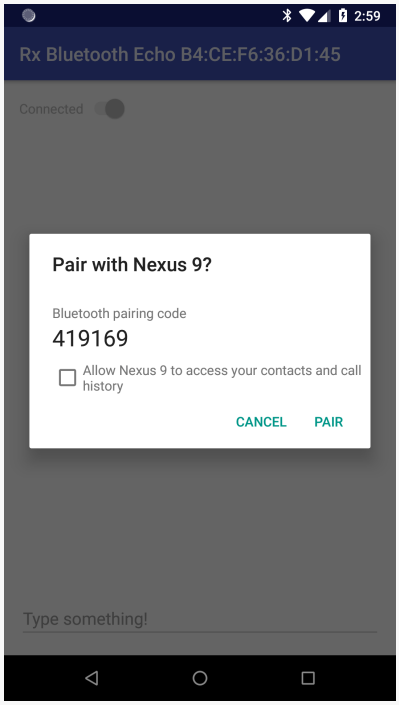


Figure 886: RxEcho Client, Showing the Pairing Dialog

Click “Pair” on both devices, though you do not need to grant rights for accessing contacts via the checkbox on the dialog.

At this point, the “Connected” switch should be in the “on” state. More importantly, the field at the bottom of the screen should be enabled. Tap on the field, type in a short message, and click the “send” action button. You should see an entry appear on the screen with your message echoed back to you in all caps:

Figure 887: RxEcho Client, Showing the Echoed Response From the Server

Your message was sent via Bluetooth RFCOMM to the server, which sent back the all-caps echo.

If you type in other messages, they will be added to the list. You can disconnect at any time, either by clicking the "Connected" switch again to toggle it off, or by pressing BACK to return to the list of device MAC addresses.

Once you have paired a device, you no longer need to discover it. So, for example, if you exit the sample app on the client and terminate its process (e.g., press the square "stop" toolbar button in Android Studio), then start it again, your server device should appear in the list immediately, even though the 120-second discoverability window may have elapsed and you have not asked the client to discover devices in any case. Your devices are paired, and you will see that pairing not only in the app on both devices, but in the Bluetooth screen in Settings as well.

The Code

The app consists of a single activity, called `MainActivity`. It uses two fragments:

1. `RosterFragment` for showing the paired and discovered devices in a `RecyclerView`
2. `DeviceFragment` for showing the "Connected" `Switch`, `EditText` for message entry, and `RecyclerView` for the responses

There is also `ShoutingEchoService`, which is a foreground `Service` that handles our incoming requests and their associated responses.

The Bluetooth logic is wrapped up in RxBluetooth. As the name suggests, this puts an RxJava wrapper around many of the Bluetooth APIs, to make it easier for us to handle thread management. The prose in this chapter will cover both the Android SDK classes and methods plus RxBluetooth; the sample just uses RxBluetooth.

Bluetooth and Permissions

A Bluetooth app usually will need three permissions:

- `BLUETOOTH`, for general Bluetooth operations
- `BLUETOOTH_ADMIN`, for Bluetooth discovery operations... which nearly every Bluetooth app will need
- either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`, in order to be able to get the Bluetooth MAC addresses of discovered or paired devices

The latter requirement is for privacy:

- Apps cannot use Bluetooth discovery and MAC addresses to determine that two users are nearby without user permission
- Apps cannot build a database of MAC addresses and physical locations, then use such a database to guess the user's location given a nearby MAC address, without user permission

`BLUETOOTH` and `BLUETOOTH_ADMIN` permissions have a `protectionLevel` of `normal`, and so all you need to do is request them in the manifest. The location permissions are `dangerous`, though, and so on Android 6.0+ devices, with a `targetSdkVersion` of 23 or higher, you need to request those at runtime. The RxBluetooth library has the `<uses-permission>` elements for `BLUETOOTH`, `BLUETOOTH_ADMIN`, and `ACCESS_COARSE_LOCATION`, which is all that we need, and therefore we do not need our own manifest entries for those.

However, we do need to request `ACCESS_COARSE_LOCATION` at runtime. Hence, this app uses the same `AbstractPermissionActivity` seen elsewhere in the book, to manage our permission request on first run of the app. `MainActivity` declares that it needs `ACCESS_COARSE_LOCATION` and exits with a `Toast` if the user declines to grant the permission.

The Rx for Your Bluetooth

RxBluetooth is available as a `com.github.ivbaranov:rxbluetooth2` artifact, which we pull in using our `dependencies` closure:

```
apply plugin: 'com.android.application'

android {
  compileSdkVersion 27

  defaultConfig {
    applicationId "com.commonsware.android.bluetooth.rxecho"
    minSdkVersion 21
    targetSdkVersion 27
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
  }

  buildTypes {
    release {
      minifyEnabled false
      proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
  }

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }

  dataBinding {
    enabled = true
  }
}

def supportVer="27.0.2"

dependencies {
  implementation("com.android.support:support-v4:$supportVer") {
    exclude group: 'com.android.support', module: 'support-media-compat'
  } // for https://issuetracker.google.com/issues/64909326
  implementation "com.android.support:recyclerview-v7:$supportVer"
  implementation "com.android.support.constraint:constraint-layout:1.0.2"
  implementation "com.github.ivbaranov:rxbluetooth2:2.0.1"
  implementation "io.reactivex.rxjava2:rxjava:2.1.7"
  implementation "io.reactivex.rxjava2:rxandroid:2.0.1"
  implementation "com.github.davidmoten:rxjava2-extras:0.1.18"
```

```
  implementation "android.arch.lifecycle:extensions:1.0.0"
}
```

(from Bluetooth/RxEcho/app/build.gradle)

We will discuss some of those other dependencies later in this chapter.

The entry point to the RxBluetooth library is an `RxBluetooth` class. Its constructor takes a `Context`, which `RxBluetooth` holds onto directly. As a result, it is safest to pass in the `Application` singleton, so we do not wind up with memory leaks. Both fragments and the `ShoutingEchoService` need to use Bluetooth, so each have their own `RxBluetooth` instances, held in `rxBluetooth` fields, initialized as those fragments and service are created, such as:

```
    rxBluetooth=new RxBluetooth(getActivity().getApplicationContext());
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

I Can Haz Bluetooth?

Roughly speaking, there are three possibilities with respect to Bluetooth on the device:

- The device lacks Bluetooth hardware
- The device has Bluetooth hardware, but it is disabled by the user
- The device has Bluetooth hardware that is ready for us to use

The first scenario should not happen. Our request of Bluetooth permissions triggers an implicit requirement of Bluetooth hardware. We would have to have `<uses-feature android:name="android.hardware.bluetooth" android:required="false" />` in our manifest to say that Bluetooth is *not* required, if we were willing to work without it. As it stands, our app should not be installable on devices that lack Bluetooth.

That being said, for illustration purposes, the sample code checks to see whether we have Bluetooth hardware, by calling `isBluetoothAvailable()` on the `RxBluetooth` object:

```
    if (!rxBluetooth.isBluetoothAvailable()) {
      Toast.makeText(getActivity(), R.string.msg_no_bt, Toast.LENGTH_LONG).show();
    }
    else {
      enableBluetooth(false);
    }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

Under the covers, RxBluetooth works with `BluetoothAdapter`, the entry point into the Bluetooth APIs in the Android SDK. `BluetoothAdapter` has a `getDefaultAdapter()` method that returns its singleton instance. If that returns `null` or has an empty address, then we do not have Bluetooth hardware.

If `isBluetoothAvailable()` returns `true`, we then go and see if Bluetooth is enabled, via an `enableBluetooth()` method. There are three possibilities that this method needs to handle:

1. Bluetooth is enabled, in which case we can start using it
2. Bluetooth is not enabled, and so we want to ask the user to enable it
3. Bluetooth is not enabled, we asked the user to enable it, and the user declined to do so

We need to handle that latter case so that we can exit gracefully, rather than continuously popping up demands that the user enable Bluetooth.

The `boolean` parameter to `enableBluetooth()` indicates whether this is the first call (as we are starting up) or the second call (after we asked the user to enable Bluetooth). The code snippet shown above — from the `onViewCreated()` method of `RosterFragment` — has the first call. `enableBluetooth()`, then, takes over:

```
void enableBluetooth(boolean didWeAskAlready) {
  if (rxBluetooth.isBluetoothEnabled()) {
    bluetoothReady();
  }
  else if (isThing()) {
    rxBluetooth.enable();

    subs.add(rxBluetooth.observeBluetoothState()
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .filter(state -> (BluetoothAdapter.STATE_CONNECTED==state))
      .subscribe(state -> bluetoothReady()));
  }
  else if (didWeAskAlready) {
    Toast.makeText(getActivity(), R.string.msg_away, Toast.LENGTH_LONG).show();
    getActivity().finish();
  }
  else {
    rxBluetooth.enableBluetooth(getActivity(), MainActivity.REQUEST_ENABLE_BLUETOOTH);
  }
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

We first call `isBluetoothEnabled()` on the `RxBluetooth` instance. That just calls `isEnabled()` on the `BluetoothAdapter`, and returns a simple `boolean` status. If Bluetooth is enabled, we call a `bluetoothReady()` method to get things started — we will examine this method in detail shortly.

If Bluetooth is not enabled, and we are running on an Android Things device, we will handle Bluetooth a bit differently, which we will examine later in this chapter.

If Bluetooth is not enabled, and we are not on an Android Things devices, *and* this is the first call (`didWeAskAlready` is `false`), we call `enableBluetooth()` on the `RxBluetooth` instance. This, in turn, will call `startActivityForResult()`, for an `Intent` with the `BluetoothAdapter.ACTION_REQUEST_ENABLE` action string. This will cause the system to display a dialog-themed activity prompting the user to enable Bluetooth. So, we pass an `Activity` and a request code into `enableBluetooth()`, and those are used to make the `startActivityForResult()` call.

The catch is that `enableBluetooth()` takes an `Activity` as a parameter and calls `startActivityForResult()` on it. Hence, the result goes to the `Activity`. This code is in `RosterFragment`, and so we cannot implement `onActivityResult()` here, as that will not be used. Instead, `onActivityResult()` goes on `MainActivity`, which then turns around and calls `enableBluetooth()` again, this time with a `true` parameter:

```
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
  if (requestCode==REQUEST_ENABLE_BLUETOOTH && roster!=null) {
    roster.enableBluetooth(true);
  }
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/MainActivity.java)

Here, `roster` is the `RosterFragment` instance, set up in `onCreate()`.

Back in `enableBluetooth()` in `RosterFragment`, if Bluetooth is not available, and we already asked the user — we are being called again by that `onActivityResult()` from `MainActivity` — we show a `Toast` and exit, since the user does not want to enable Bluetooth.

Ideally, eventually, the user enables Bluetooth, so our app can work.

I Feel a Bond Between Us

Next, we need to provide the user with a list of devices that might be running our app. This list comes from two sources:

1. The list of Bluetooth devices paired with this one
2. The list of Bluetooth devices discovered by our app

However, this list will also need to be filtered, so we do not present clearly silly options to the user, such as suggesting that our app is running on a Bluetooth-connected speaker.

Getting the Paired Devices

`BluetoothAdapter` has a `getBondedDevices()` method; `RxBluetooth` wraps that in its own `getBondedDevices()` method. This returns the list of "bonded" devices, which pretty much everyone else would refer to as the "paired" devices.

Specifically, we get back a list of `BluetoothDevice` objects. We will need those objects later on for connecting to and communicating with the other device, so we will maintain a model collection of these objects.

The `bluetoothReady()` method is called when we know that Bluetooth is ready for use. There, part of its work is to call `initAdapter()`:

```
private void initAdapter() {
  adapter.setItems(rxBluetooth.getBondedDevices());
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

Here, we call `getBondedDevices()` and pass that to a `setItems()` method on an `adapter`. `adapter` is a `DevicesAdapter` instance that we set up when we create the fragment. `DevicesAdapter` is a `RecyclerView.Adapter`, tied to a `RowHolder` for our list rows:

```
private class DevicesAdapter extends RecyclerView.Adapter<RowHolder> {
  private final ArrayList<BluetoothDevice> devices=new ArrayList<>();

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowHolder(RosterRowBinding.inflate(getLayoutInflater(), parent, false)));
  }
```

```
  @Override
  public void onBindViewHolder(RowHolder holder, int position) {
    holder.bind(devices.get(position));
  }

  @Override
  public int getItemCount() {
    return(devices.size());
  }

  void setItems(Collection<BluetoothDevice> devices) {
    this.devices.clear();

    for (BluetoothDevice device : devices) {
      if (isCandidateDevice(device)) {
        this.devices.add(device);
      }
    }

    notifyDataSetChanged();
  }

  void addDevice(BluetoothDevice device) {
    if (isCandidateDevice(device) && !devices.contains(device)) {
      devices.add(device);
      notifyItemInserted(devices.size()-1);
    }
  }

  boolean isCandidateDevice(BluetoothDevice device) {
    int deviceClass=device.getBluetoothClass().getDeviceClass();

    return(((deviceClass & BluetoothClass.Device.Major.COMPUTER)==
      BluetoothClass.Device.Major.COMPUTER) ||
      ((deviceClass & BluetoothClass.Device.Major.PHONE)==
      BluetoothClass.Device.Major.PHONE) ||
      ((deviceClass & BluetoothClass.Device.Major.AUDIO_VIDEO)==
        BluetoothClass.Device.Major.AUDIO_VIDEO));
  }
}

public class RowHolder extends RecyclerView.ViewHolder {
  private final RosterRowBinding binding;

  RowHolder(RosterRowBinding binding) {
    super(binding.getRoot());

    this.binding=binding;
  }

  void bind(BluetoothDevice device) {
    binding.setDevice(device);
    binding.setController(this);
    binding.executePendingBindings();
  }

  public void onClick(BluetoothDevice device) {
    ((Contract)(getActivity())).showDevice(device);
  }
```

```
  }
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

Filtering the Devices

Most of the above code follows from what we saw with `RecyclerView` and the data binding framework elsewhere in the book. What's a little strange is the filtering going on in methods like the `setItems()` that `initAdapter()` calls.

Bluetooth devices have classes. Computers are a different class than are audio devices, which are in a different class than are toys, and so on. An Android device might discover, or be paired with, other devices from a variety of classes. Rather than present all possible devices, it makes sense to limit the devices to things that are reasonably likely to have our app.

`BluetoothDevice` has a `getBluetoothClass()` method, which returns a `BluetoothClass` object. That, in turn, has a `getDeviceClass()` method, which returns an `int` that represents the device class.

There are three classes (at least) that Android devices can fall in, identified by constants defined on the `BluetoothClass.Device.Major` class:

- Some phones and tablets report themselves as being a `COMPUTER`
- Some phones report themselves as being a `PHONE`
- An Android Things device inexplicably reports itself as being `AUDIO_VIDEO`

So, `isCandidateDevice()` does some bit-checking to see if the device we are bonded to is in one of those three major classes, rejecting any others.

Listing the Devices

The `onViewCreated()` method of `RosterFragment` sets up that `RecyclerView`, as well as creating the `RxBluetooth` instance, seeing if we have Bluetooth hardware, and triggering the initial call to `enableBluetooth()`:

```
@Override
public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  rv=view.findViewById(R.id.devices);

  rv.setLayoutManager(new LinearLayoutManager(getActivity()));
  rv.addItemDecoration(new DividerItemDecoration(getActivity(),
```

```
      LinearLayoutManager.VERTICAL));
    rv.setAdapter(adapter);

    rxBluetooth=new RxBluetooth(getActivity().getApplicationContext());

    if (!rxBluetooth.isBluetoothAvailable()) {
      Toast.makeText(getActivity(), R.string.msg_no_bt, Toast.LENGTH_LONG).show();
    }
    else {
      enableBluetooth(false);
    }
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

A Voyage of Discovery

Before we can connect to a Bluetooth device, we need to pair with it. And before we can pair with it, we need to discover it. So, while the preceding section outlined how we can show the user already-paired devices, we also have to allow the user to discover devices to pair with.

This is a three-part process:

1. One device — the server — has to become discoverable, as devices are not discoverable by default, for security reasons
2. The other device — the client — has to scan for discoverable devices
3. If and when any devices are discovered, we need to add them to our list of devices to potentially connect to

Enabling Discoverability

In the Android SDK, to allow other devices to discover the one your app is running on, you use `startActivityForResult()` on a `BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE` `Intent`. This will pop up a system-supplied dialog-themed activity to confirm that the user really wants the device to be discoverable. If the user agrees, then the device will be discoverable... for a short while.

The default discoverability duration in 120 seconds. You can add a `BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION` extra to the `Intent`, with a value in seconds (up to 300) of how long you would like the device to be discoverable.

RxBluetooth hides all of this behind a pair of `enableDiscoverability()` methods. Both take an `Activity` and a request code, for use with the

`startActivityForResult()` call. One of the `enableDiscoverability()` methods also takes a duration, to pass along via `EXTRA_DISCOVERABLE_DURATION`.

As with `enableBluetooth()`, the `startActivityForResult()` call is made on the supplied `Activity`, not any fragment. If you want the result, you will need to implement `onActivityResult()` on the same `Activity` that you supplied to the `enableDiscoverability()` call. Here, the result is contained entirely in the result code, which will be `RESULT_CANCELED` if the user declined to enable discoverability... or the duration in seconds that the device will be discoverable. `RESULT_OK` is *not* the value indicating success in this case.

In `RosterFragment`, if the user taps on the "Allow Discovery" action bar item, we just call `enableDiscoverability()`:

```
        rxBluetooth.enableDiscoverability(getActivity(), REQUEST_ENABLE_DISCOVERY);
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

In our case, we are ignoring the result, as we have no behavior to invoke based upon that result.

There is no way to cancel discoverability — it will simply time out on its own.

Discovering Other Devices

In the Android SDK, `BluetoothAdapter` has three methods tied to discovery (the act of discovering discoverable devices):

- `startDiscovery()`
- `isDiscovering()`
- `cancelDiscovery()`

RxBluetooth has its own versions of those methods, which just call the corresponding method on the wrapped `BluetoothAdapter`.

The three methods do pretty much what you would expect from their names: start searching for discoverable devices, report if a discovery search is ongoing, and stop the search.

In `RosterFragment`, if the user taps the "Discover" action bar item, we just call `startDiscovery()`:

```
        rxBluetooth.startDiscovery();
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

As part of cleanup, we call `cancelDiscovery()` in `onDestroy()`, so a discovery operation will not continue past the lifetime of the fragment instance.

Reacting to Discovery Results

However, somewhere along the line, it would be very useful if we actually found out about the discovery results. If the scan finds devices, we need to know about them. We also need to know when the discovery scan stops of its own accord, as it will not run forever.

To that end, as part of our work in `bluetoothReady()` in `RosterFragment`, we set up two RxJava chains. One will use `observeDevices()` on `RxBluetooth` to find out about discovered devices. The other will use `observeDiscovery()` on `RxBluetooth` to find out about the status of the discovery scan itself:

```
      subs.add(rxBluetooth.observeDevices()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(this::addDevice));

      subs.add(rxBluetooth.observeDiscovery()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(s ->
          discover.setEnabled(!BluetoothAdapter.ACTION_DISCOVERY_STARTED.equals(s))));
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

`observeDiscovery()`, under the covers, uses a `BroadcastReceiver` to listen for `BluetoothAdapter.ACTION_DISCOVERY_STARTED` and `BluetoothAdapter.ACTION_DISCOVERY_FINISHED` broadcasts, forwarding them along to our chain. We simply enable and disable the "Discovery" `MenuItem` depending upon the scan state.

Similarly, `observeDevices()` uses a `BroadcastReceiver` for `BluetoothDevice.ACTION_FOUND` devices, forwarding the `BluetoothDevice` extra (`EXTRA_DEVICE`) along to our chain. We call `addDevice()`, which will add the device to our `RecyclerView`, if that device is not already in the list. It might already be in the list because:

- We were paired to it, but it happened to be discoverable during our scan, or

- We had it in the list from a previous scan

Serving and Shouting

Given all of that, our client device can find our server device. Now, we need to set up the actual communications between them.

First, let's look at how our server listens for incoming connections from clients. That is managed by the `ShoutingEchoService`, so that we can listen for connections and respond to requests from the background, if desired.

Service Scaffolding

`ShoutingEchoService` is a foreground service, partly so the user can stop the service from the `Notification`, and partly so that the service can run for an extended period of time. So, in `onCreate()`, among other things, we set up our notification channel and call `startForeground()`:

```
    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    startForeground(1338, buildForegroundNotification());
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

`startForeground()` relies on a `buildForegroundNotification()` method to create the `Notification`:

```
private Notification buildForegroundNotification() {
  NotificationCompat.Builder b=
    new NotificationCompat.Builder(this, CHANNEL_WHATEVER);

  b.setOngoing(true)
    .setContentTitle(getString(R.string.msg_foreground))
    .setSmallIcon(R.drawable.ic_stat_ping)
    .addAction(android.R.drawable.ic_media_pause, getString(R.string.msg_stop),
      buildStopPendingIntent());

  return(b.build());
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

`buildForegroundNotification()`, in turn, calls `buildStopPendingIntent()`, to craft a `PendingIntent` to be tied to the "stop" action added to the `Notification`. `buildStopPendingIntent()` uses a service `PendingIntent`, wrapped around an explicit `Intent` (identifying our service) to which we also happen to attach a custom action string (`ACTION_STOP`):

```
private PendingIntent buildStopPendingIntent() {
  Intent i=new Intent(this, getClass()).setAction(ACTION_STOP);

  return(PendingIntent.getService(this, 0, i, 0));
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

`onStartCommand()` then examines the `Intent` used to start the service, sees if it is `ACTION_STOP`, and calls `stopSelf()` if so:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
  if (ACTION_STOP.equals(intent.getAction())) {
    stopSelf();
  }

  return(START_NOT_STICKY);
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

As a result, if the user taps the "Stop Server" notification action, the service will stop itself.

A Quick Word About MutableLiveData

The `RosterFragment` has a checkable action bar item, "Server", used to start and stop the `ShoutingEchoService`. To make that work, the `RosterFragment` needs to know about the status of the service, and *changes* to the status of the service. For example, if the user stops the service through the notification action, we need to uncheck the `MenuItem` if our UI is still around.

We could use an event bus, such as greenrobot's EventBus. In this case, we are using `MutableLiveData`.

`MutableLiveData` is from the Architecture Components family of libraries. `LiveData` is a bit like a very tiny subset of RxJava, allowing you to observe on a stream of data. `MutableLiveData` is a subclass of `LiveData` that allows external parties to post events to observers. As a result, `MutableLiveData` can be used a bit like an event bus. The main "claim to fame" for `LiveData` — beyond it coming from Google — is that its mechanism for observing changes is lifecycle-aware, so we do not need to register and unregister from activity or fragment lifecycle methods.

You can learn a lot more about `MutableLiveData`, `LiveData`, and the rest of the Architecture Components in the companion volume, *Android's Architecture Components*.

Keeping the UI in the Loop

To keep the UI layer informed about the status of the service, `ShoutingEchoService` has a `MutableLiveData` that publishes `Status` objects:

```
  static final MutableLiveData<Status> STATUS=new MutableLiveData<>();
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

Here, `Status` is a simple wrapper around a `boolean`:

```
  static class Status {
    static final Status IS_RUNNING=new Status(true);
    static final Status NOT_RUNNING=new Status(false);
    final boolean isRunning;

    private Status(boolean isRunning) {
      this.isRunning=isRunning;
    }
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

This could be replaced by an `enum` or `Boolean`, but you might have more complex data that you want to publish as part of the status.

Then, in `onCreate()`, we use `postValue()` on `MutableLiveData` to publish that the service is running:

```
  @Override
  public void onCreate() {
    super.onCreate();
```

```
    rxBluetooth=new RxBluetooth(getApplicationContext());
    acceptConnections();

    NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      mgr.getNotificationChannel(CHANNEL_WHATEVER)==null) {
      mgr.createNotificationChannel(new NotificationChannel(CHANNEL_WHATEVER,
        "Whatever", NotificationManager.IMPORTANCE_DEFAULT));
    }

    startForeground(1338, buildForegroundNotification());

    STATUS.postValue(Status.IS_RUNNING);
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

In `onDestroy()`, among other things, we publish that the service is no longer running:

```
    STATUS.postValue(Status.NOT_RUNNING);
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

`RosterFragment` sets up a lambda expression to observe these status changes and update the checked state of a `server` `MenuItem`:

```
    ShoutingEchoService.STATUS.observe(this,
      status -> {
        if (server!=null && status!=null) server.setChecked(status.isRunning);
      });
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

That informs us in real time about status *changes*. To find out the current status, we have to call `getValue()` on the `MutableLiveData`, which `RosterFragment` wraps in an `isServerRunning()` method:

```
  private boolean isServerRunning() {
    ShoutingEchoService.Status status=ShoutingEchoService.STATUS.getValue();

    return(status!=null && status.isRunning);
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

That can then be used to control what happens when the user taps that "Server" action bar item:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.server:
      if (isServerRunning()) {
        getActivity().stopService(new Intent(getActivity(),
          ShoutingEchoService.class));
      }
      else {
        getActivity().startService(new Intent(getActivity(),
          ShoutingEchoService.class));
      }

      return(true);

    case R.id.discover:
      rxBluetooth.startDiscovery();
      return(true);

    case R.id.allow_disco:
      rxBluetooth.enableDiscoverability(getActivity(), REQUEST_ENABLE_DISCOVERY);
      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

Services with Services

Of course, none of this has anything much to do with Bluetooth. It would be nice if our server actually used Bluetooth somewhere.

Up in `onCreate()`, in addition to initializing our `RxBluetooth` instance, we call `acceptConnections()`. That is where the Bluetooth fun starts with our service:

```
private void acceptConnections() {
  connectionSub=rxBluetooth
    .observeBluetoothSocket(getString(R.string.app_name), SERVICE_ID)
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe(this::operateServer,
      throwable -> Log.e(getClass().getSimpleName(),
        "Exception from Bluetooth", throwable));
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

In the Android SDK, a server can start listening for RFCOMM connection requests by calling `listenUsingRfcommWithServiceRecord()` on the `BluetoothAdapter` instance. That is wrapped by `observeBluetoothSocket()` on `RxBluetooth`, so we can observe the results of that work.

`listenUsingRfcommWithServiceRecord()` takes two parameters:

- Some sort of display name, for unknown reasons
- A UUID identifying the Bluetooth service that we are publishing

Clients that try to connect to us will need to use that same UUID, to distinguish our Bluetooth service from other ones. In our case, the UUID is `SERVICE_ID`:

```
  static final UUID SERVICE_ID=
      UUID.fromString("20c6de08-2cf5-4ca2-96af-cb0a45055d37");
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

`listenUsingRfcommWithServiceRecord()` returns a `BluetoothServerSocket`. Calling `accept()` on it blocks until a connection is made, at which time `accept()` returns a `BluetoothSocket` that can be used for communicating with that connected client. At this point, you `close()` the `BluetoothServerSocket`, as a given instance of `BluetoothServerSocket` is used for establishing just one connection. All of that work is wrapped up inside of RxBluetooth, so we can just set up an RxJava chain, using background threads both for listening for incoming connections and consuming the resulting `BluetoothSocket`.

Once we get a `BluetoothSocket`, we call `operateServer()`. Among other things, `operateServer()` turns around and calls `acceptConnections()` again, so we can accept the next incoming client connection request. So that we do not wind up with a bunch of RxJava subscriptions piling up, we keep track of the outstanding one in a `connectionSub` field, disposing of it both in `onDestroy()` (as we no longer need it then) and in `acceptConnections()` (as we no longer need the previous subscription, since we already received our `BluetoothSocket`).

Reach Out and Touch Someone

Given that our server has its service to respond to connections, our client has to make those connections.

On the client side, this is managed by the `DeviceFragment`. We create and show one of these when the user clicks on a device in the list displayed in the `RosterFragment`. As it turns out, `BluetoothDevice` is `Parcelable`, so we can pass that over to the `DeviceFragment` using the arguments `Bundle`:

```
  public static Fragment newInstance(BluetoothDevice device) {
    DeviceFragment result=new DeviceFragment();
    Bundle args=new Bundle();

    args.putParcelable(ARG_DEVICE, device);
    result.setArguments(args);

    return(result);
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

```
  private BluetoothDevice getDevice() {
    return(getArguments().getParcelable(ARG_DEVICE));
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

Finding Out When We Should Connect

The `DeviceFragment` UI is mediated by the data binding framework. The fragment's layout — `res/layout/device.xml` — injects the fragment itself as a controller and has the `Switch` invoke `onConnectionChange()` on the fragment when the `Switch` state changes:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <data>

    <variable
      name="controller"
      type="com.commonsware.android.bluetooth.rxecho.DeviceFragment" />
  </data>

  <android.support.constraint.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/padding_main">
```

```
    <Switch
      android:id="@+id/connected"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onCheckedChanged="@{() -> controller.onConnectionChange()}"
      android:text="@string/switch_connected"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent" />

    <android.support.v7.widget.RecyclerView
      android:id="@+id/transcript"
      android:layout_width="0dp"
      android:layout_height="0dp"
      app:layoutManager="LinearLayoutManager"
      app:layout_constraintBottom_toTopOf="@+id/entry"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toBottomOf="@+id/connected"
      app:stackFromEnd="true" />

    <EditText
      android:id="@+id/entry"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:ems="10"
      android:enabled="false"
      android:hint="@string/hint_entry"
      android:inputType="text"
      android:imeOptions="actionSend"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintHorizontal_bias="0.627"
      app:layout_constraintStart_toStartOf="parent" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

(from Bluetooth/RxEcho/app/src/main/res/layout/device.xml)

`onCreateView()` then sets up the `DeviceBinding`, holding onto it in a `binding` field:

```
  @Nullable
  @Override
  public View onCreateView(LayoutInflater inflater,
                           @Nullable ViewGroup container,
                           @Nullable Bundle savedInstanceState) {
    binding=DeviceBinding.inflate(inflater, container, false);
    binding.setController(this);
```

```
    return(binding.getRoot());
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

Connecting and Disconnecting

That `onConnectionChange()` method then checks to see if the `Switch` is checked. If it is, we disable the `Switch`, so the disabled state indicates that we are in a "connecting" state, as opposed to a "disconnected" or "connected" state:

```
public void onConnectionChange() {
  if (binding.connected.isChecked()) {
    binding.connected.setEnabled(false);
    connectionSub=rxBluetooth.observeConnectDevice(getDevice(), ShoutingEchoService.SERVICE_ID)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(this::onConnected, this::onConnectionError);
  }
  else {
    disconnect();
  }
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

More importantly, we set up an RxJava chain based on `observeConnectDevice()` on our `RxBluetooth` object. This will try to connect to our service, given the `BluetoothDevice` of the server's device and the UUID of the service.

`observeConnectDevice()` wraps a call to `createRfcommSocketToServiceRecord()` on `BluetoothDevice`. This creates a `BluetoothSocket` representing the client side of the connection to the UUID-identified service running on the designated device. A call to `connect()` will block until we successfully connect to that service or it throws some form of `IOException` indicating that we could not connect.

Our RxJava chain routes successful connections to an `onConnected()` method. Among other things, it enables the `Switch` (to show that we are connected) and the `EditText` (initially disabled, since we cannot send messages until we are connected). The rest has to do with data exchange, which we will look at in the next section.

Our RxJava chain routes exceptions to `onConnectionError()`, which unchecks the `Switch` (since we are not connected), logs the details to LogCat (in case this error was unexpected), and shows a `Toast`.

The fact that we uncheck the `Switch` triggers a fresh call to `onConnectionChange()`. If the `Switch` is unchecked, we call a `disconnect()` method, which, among other things, will `dispose()` the `connectionSub` subscription, as we no longer need that RxJava chain for our failed connection attempt.

There is nothing in this code specific to pairing. That happens automatically when we try to connect to the service on the other device. If we are already paired, getting our `BluetoothSocket` is relatively quick. If we are not paired, we need to wait for the user to agree to pair the devices, which may take some time or possibly never happen (e.g., user declines, user fails to notice the dialog and it times out).

Ping and Pong

Once our client has connected to the server, we can start exchanging messages.

Technically, there is nothing in Bluetooth that requires a request/response protocol. Both sides can communicate as needed to the other. However, a request/response pattern is easy to write, which is why we use it here.

Getting Our Client Streams

A `BluetoothSocket` has `getInputStream()` and `getOutputStream()` methods for receiving and sending data to the other party, respectively. RxBluetooth does not add anything for these — you are on your own for determining when and how to use those streams.

The `onConnected()` method, in addition to adjusting the state of some widgets:

- Holds onto our `BluetoothSocket`
- Create a `PrintWriter` tied to the output stream from the `BluetoothSocket`
- Arranges to process the data arriving on the input stream using an RxJava chain

```
private void onConnected(BluetoothSocket socket) throws IOException {
  binding.connected.setEnabled(true);
  binding.entry.setEnabled(true);
  this.socket=socket;
  out=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
  responseSub=Bytes.from(socket.getInputStream())
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
```

```
      .subscribe(bytes -> post(new String(bytes)),
        throwable -> out.close());
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

We will examine that RxJava chain in greater detail a bit later in this chapter.

Sending the Message

The `DeviceFragment` layout, shown above, has `android:imeOptions="actionSend"`, to ask for a "send" button on any soft keyboard. In `onViewCreated()`, we tie the `EditText` to a `send()` method by means of a lambda expression supplied to `setOnEditorActionListener()`, to find out when the user taps that "send" button:

```
@Override
public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  binding.entry.setOnEditorActionListener((v, actionId, event) -> (send()));
  binding.entry.setEnabled(socket!=null);

  RecyclerView rv=view.findViewById(R.id.transcript);

  rv.setAdapter(adapter);
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

`send()`, in turn, uses the `PrintWriter` to send the entered text to the server:

```
public boolean send() {
  Single.just(binding.entry.getText().toString())
    .observeOn(Schedulers.io())
    .subscribe(message -> {
      out.print(message);
      out.flush();
    });

  return(true);
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

We want to do that work on a background thread, as we always want to do I/O on a background thread. To stick to a single thread pool, we wrap the text that the user

types in into a `Single` (a "one-shot" RxJava type otherwise reminiscent of `Observable`), observe the data on the `io()` thread, and in there `print()` the text to the `PrintWriter` and `flush()` it to make sure that it goes out.

Processing the Request

Back in `ShoutingEchoService`, the `operateServer()` method gets the `BluetoothSocket` representing the server side of the connection and sets up its own RxJava chain to work with it:

```
private void operateServer(BluetoothSocket socket) throws IOException {
  disconnect();
  this.socket=socket;
  acceptConnections();

  final PrintWriter out=
    new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));

  Bytes.from(socket.getInputStream())
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.computation())
    .subscribe(bytes -> {
      out.print(new String(bytes).toUpperCase());
      out.flush();
    }, throwable -> out.close());
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

First, though, we call `disconnect()`, which closes any existing `BluetoothSocket` that we might have:

```
private void disconnect() {
  if (socket!=null) {
    try {
      socket.close();
    }
    catch (IOException e) {
      Log.e(TAG, "Exception from Bluetooth", e);
    }
  }
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/ShoutingEchoService.java)

As noted above, we also call `acceptConnections()`, so we can accept a connection from the next client. We only support one client at a time, closing the `BluetoothSocket` from the previous client as part of accepting the next one here. That's mostly a limitation of the sample app, to try to keep this sample from being excessively complex.

RxJava does not provide anything that works directly with Java's stream classes, like `InputStream` or `OutputStream`. However, David Moten offers the `rxjava2-extras` library which adds more Java-specific bridges to RxJava. Here, we use the `Bytes` class, which sets up RxJava chains for data coming in from an `InputStream`.

We want to receive the message on a background thread, and we also want to send the response on a background thread. To prevent sending the response from blocking future input, we use separate threads, with the `io()` thread for reading the data off of the `InputStream` and the `computation()` thread for sending the response.

Actually sending the response then is simply a matter of:

- Converting the bytes into a `String`
- Capitalizing the `String` using `toUpperCase()`, so the service appears to be shouting
- Using `print()` and `flush()` on our `PrintWriter` to send the response back to the client

If there is an `IOException`, we just close the `PrintWriter`, which closes the associated `OutputStream`. `Bytes` automatically closes our `InputStream` when the stream is closed from the sending side.

Receiving the Response

In `DeviceFragment`, part of what `onConnected()` does is set up a similar `Bytes`-based RxJava chain to listen for the responses from the server:

```
private void onConnected(BluetoothSocket socket) throws IOException {
  binding.connected.setEnabled(true);
  binding.entry.setEnabled(true);
  this.socket=socket;
  out=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
  responseSub=Bytes.from(socket.getInputStream())
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(bytes -> post(new String(bytes)),
```

```
        throwable -> out.close());
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

Here, we get the data on the main application thread, using `AndroidSchedulers.mainThread()`, so we can take the bytes, convert them into a `String`, and call a `post()` method. `post()`, in turn, clears out the `EditText` (to prepare for the next message) and adds the response message to a `TranscriptAdapter` that populates the `RecyclerView` that dominates this fragment's UI:

```
  private void post(String message) {
    binding.entry.setText("");
    adapter.add(message);
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/DeviceFragment.java)

The result is that when we get the response, it appears in the list of responses, reminiscent of a chat client.

Differences with Android Things

There are a couple of differences in how the RxEcho app runs on an Android Things platform.

First, we need to know that we *are* on an Android Things platform. To determine that, we have an `isThing()` method that checks to see if we have the `FEATURE_EMBEDDED` system feature or not:

```
  private boolean isThing() {
    return(Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&
      (getActivity().getPackageManager().hasSystemFeature(PackageManager.FEATURE_EMBEDDED)));
  }
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

In `enableBluetooth()`, if we are on an Android Thing, and we do not already have Bluetooth enabled, we can enable it directly ourselves, by calling `enable()` on the `RxBluetooth` instance (which, in turn, calls `enable()` on the `BluetoothAdapter`). This is frowned upon in conventional Android apps, where we should ask the user if it is OK to enable Bluetooth. In the case of an Android Thing, we have no way to ask

the user — not only might the Thing not have a screen, but the system-supplied confirmation activity does not even exist.

However, `enable()` is asynchronous. We get control right away, and Bluetooth will not yet be powered on. We need to find out when it is ready, so we can continue. `RxBluetooth` offers an `observeBluetoothState()` method to allow us to set up an RxJava chain to find out about state changes in Bluetooth itself. As with some of the other `observe...()` methods, this wraps around a `BroadcastReceiver`, in this case one that watches for `BluetoothAdapter.ACTION_STATE_CHANGED` events, passing to us the state as an `Integer`. There are a few constants out on `BluetoothAdapter` for different possible states; the one that we want is `STATE_CONNECTED`. We use an RxJava `filter()` operator to only pass along `STATE_CONNECTED` to our subscriber, which then calls `bluetoothReady()` to set up the rest of the Bluetooth support.

`bluetoothReady()` itself behaves substantially differently depending on whether we are on a Thing or not:

```
private void bluetoothReady() {
  isReady=true;

  if (isThing()) {
    getActivity().startService(new Intent(getActivity(),
      ShoutingEchoService.class));
    rxBluetooth.enableDiscoverability(getActivity(), REQUEST_ENABLE_DISCOVERY,
      300);
    Toast.makeText(getActivity(), R.string.msg_disco, Toast.LENGTH_LONG).show();
  }
  else {
    updateMenu();
    initAdapter();

    subs.add(rxBluetooth.observeDevices()
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(this::addDevice));

    subs.add(rxBluetooth.observeDiscovery()
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(s ->
        discover.setEnabled(!BluetoothAdapter.ACTION_DISCOVERY_STARTED.equals(s))));
  }
}
```

(from Bluetooth/RxEcho/app/src/main/java/com/commonsware/android/bluetooth/rxecho/RosterFragment.java)

If we are not on a Thing, we:

- Make sure our action bar overflow items are enabled as appropriate
- Populate our `RecyclerView` with paired devices

- Set up the RxJava chain to add discovered devices to the list as well

On a Thing, we do not need any of that, as we do not necessarily have a screen to work with. Instead, we:

- Start the `ShoutingEchoService` directly
- Enable discoverability for 300 seconds, so there is a five-minute window in which clients can find our Thing
- Show a `Toast` about the discoverability, in case we *do* happen to have a screen (as might be the case on a Raspberry Pi)

The net is that on a Thing, we immediately set up the echo server and make it possible for clients to pair with it, since we cannot rely on the user to request those things from the action bar.

Dealing with Different Hardware

While a lot of focus is placed on screen sizes, there are many other possible hardware differences among different Android devices. For example, some have telephony features, while others do not.

There is a three-phase plan for dealing with these variations:

1. **Filter** out devices that cannot possibly run your app successfully, so your app will not appear to them in the Play Store and they will be unable to install your app if obtained by other means
2. **React** to varying hardware that you can support, but perhaps might support differently (e.g., choosing a particular flash mode for a device having a camera with a flash)
3. **Cope** with device bugs or regressions that impact your application

This chapter will go through each of these topics.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Filtering Out Devices

Elsewhere in the book, we discussed a few manifest entries that will serve to filter out devices that cannot run your app:

- android:minSdkVersion in the <uses-sdk> element, to stipulate that devices must run a certain version of Android (or higher)
- <supports-screens> and <compatible-screens>, which indicate which screens sizes and densities you are capable of supporting

This section outlines other "advertisements" that you can put in the manifest to restrict which devices run your app.

uses-feature

The <uses-feature> element restricts your app to devices that have certain hardware features. For each element, you supply the name of a feature (e.g., android.hardware.telephony) and whether or not it is required:

```
<uses-feature
  android:name="android.hardware.camera"
  android:required="false" />
```

By default, android:required is set to true, so typically you will only see it in a manifest when it is set to false.

You might wonder why we would bother ever setting android:required to false. After all, that should have the same effect as not listing it at all. In practice, though, it has two major uses.

First, markets like the Play Store might highlight the fact that you *can* use a particular hardware capability, even though you do not strictly require it.

More importantly, you can use android:required="false" to *undo* a *requirement* that Android infers from your permissions. Requesting some permissions causes Android to assume — for backwards-compatibility reasons — that your app needs the affiliated hardware. For example, requesting the CAMERA permission causes Android to assume that you need a camera (android.hardware.camera) *and* that the camera support auto-focus (android.hardware.camera.autofocus). If, however, you are requesting the permission because you would like to use the hardware if available, but can live without it, you need to expressly add a <uses-feature> element declaring that the hardware feature is not required.

For example, in February 2010, the Motorola XOOM tablet was released. This was the first Android device that had the Play Store on it and truly had no telephony capability. As such, the XOOM would be filtered out of the then-Android Market

(now Play Store) for any app that required permissions like `SEND_SMS`. Many developers requested this permission, even though their apps could survive without SMS-sending capability. However, their apps were still filtered out if they did not have the `<uses-feature>` element declaring that telephony was not required.

You can find a table listing Android permissions and assumed hardware feature requirements in the Android developer documentation.

uses-configuration

The `<uses-configuration>` element is very reminiscent of `<uses-feature>`: it dictates hardware requirements. The difference is two-fold:

1. It focuses on hardware elements that represent different device configurations, meaning that you might use different resources for them
2. It allows you to specify *combinations* of capabilities that you need

There are three capabilities that you can require via `<uses-configuration>`:

1. The existence of a five-way navigation control, whether a specific type (D-pad, trackball, etc.) or any such control
2. The existence of a physical keyboard, whether a specific type (QWERTY, 12-key numeric keypad, etc.) or any such keyboard
3. A touchscreen

You can have as many `<uses-configuration>` elements as you need – any device that matches at least one such configuration will be eligible to install your app.

For example, the following `<uses-configuration>` element restricts your app to devices that have some sort of navigation control but do not necessarily have a touchscreen, such as a Android TV device:

```
<uses-configuration
  android:reqFiveWayNav="true"
  android:reqTouchScreen="notouch" />
```

uses-library

The `<uses-library>` element tells Android that your application wishes to use a particular *firmware*-supplied library. The most common case for this was Maps V1,

which is shipped in the form of an SDK add-on and firmware library. This, however, has been deprecated for quite some time.

However, there are other firmware libraries that you might need. These will typically be manufacturer-specific libraries, allowing your application to take advantage of particular beyond-the-Android-SDK capabilities of a particular device. This is very uncommon nowadays.

The Google Play Store will filter out your application from devices that lack a firmware library that you require via `<uses-library>`. If the user tries installing your app by some other means (e.g., download from a Web site), your app will fail to install on devices that lack the firmware library.

If you conditionally want the firmware library — you will use it if available but can cope if it is not — you can add `android:required="false"` to your `<uses-library>` element. That will allow your app to install and run on devices missing the library in question. Detecting whether or not the library exists in your process at runtime is a matter if using `Class.forName()` to see if you have access to some class from that library, where a `ClassNotFoundException` means that you do not have the library.

Runtime Capability Detection

Reacting to device capabilities is the second phase of dealing with different devices. Some features you might want (e.g., telephony for sending SMSes) but can live without. Other features may have subtle variations that you cannot filter against and therefore need to adapt to at runtime (e.g., possible picture resolutions off of a camera).

This section will cover various techniques for determining what a device can do, at runtime, so you can react accordingly.

Features

Any feature you do not make required via `<uses-feature>` can be detected at runtime by calling `hasSystemFeature()` on `PackageManager`. For example, if you would like to send SMS messages, but only on telephony-capable devices, you could have the following `<uses-feature>` element:

```
<uses-feature
  android:name="android.hardware.telephony"
  android:required="false" />
```

Then, at runtime, you can call `hasSystemFeature(PackageManager.FEATURE_TELEPHONY)` on a `PackageManager` instance to find out if, indeed, the device has telephony capability and sending SMSes should work.

Other Capabilities

Various subsystems have their own means of helping you determine what is possible or not:

- The camera APIs can let you know the capabilities of a camera (e.g., whether or not it has a flash, and what specific flash modes are supported).
- The `LocationManager` will help you determine what location providers are available that meet your `Criteria`.
- The sensor subsystem lets you find out what sensors are installed, either overall or for a particular type (e.g., accelerometer).

Dealing with Device Bugs

Alas, devices are not perfect. Even though the Compatibility Test Suite attempts to ensure that all Android devices legitimately running the Play Store faithfully implement the Android SDK, some device manufacturers make changes that introduce bugs.

Just as Web developers can "sniff" on the `User-Agent` HTTP header to determine what sort of browser is requesting a page, you can use the `Build` class to determine what sort of device is running your app. If you encounter problems with a specific device, you may be able to use `Build` to identify that device at runtime and "route around the damage".

Trail: Integration and Introspection

Writing and Using Parcelables

`Parcelable` is a marker interface, reminiscent of `Serializable`, that shows up in many places in the Android SDK. `Parcelable` objects can be put into `Intent` extras or `Bundle` objects, for example. Making your own custom classes implement `Parcelable` greatly increases their flexibility.

At the same time, `Parcelable` is something that can be overused. In most Android apps, few if any custom classes really need to have `Parcelable` capabilities.

In this chapter, we will review how to modify classes to implement `Parcelable` and what the limitations are on using `Parcelable`.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

The Role of Parcelable

A `Parcelable` object is one that can be placed into a `Parcel`. A `Parcel` is the primary vehicle for passing data between processes in Android's inter-process communication (IPC) framework.

IPC abounds in Android, even in places where you may not expect it. Every time you call `startActivity()`, for example, IPC occurs, even if the activity that calls `startActivity()` and the activity to be started are in the same process. A core OS process is the one that is responsible for identifying the activity to be started and routing control to it, so `startActivity()` performs IPC from the original activity's

process to a core OS process. The core OS process then eventually performs IPC to the target process for the activity to be started.

If you see an `Intent` or a `Bundle` in the Android SDK, odds are that those objects are involved in IPC. That is not always the case — `LocalBroadcastManager`, for example, uses `Intent` objects purely in-process — but it is a reasonable rule of thumb. Hence, there is keen interest in being able to implement `Parcelable` on specific classes, either to pass to other components via `Intent` extras, or to become part of the saved instance state `Bundle`.

`Parcelable` objects are also important for use with remote services via the binding pattern.

Writing a Parcelable

You have three major approaches for adding `Parcelable` capabilities to your classes in Android:

1. Use an annotation processor that will add in the appropriate bits of magic for you
2. Use a code generator site or tool that will take your existing class as input and give you the `Parcelable`-enabled rendition as output
3. Just do it yourself

By Annotations

Enterprising developers have created annotation processing libraries that can be used to add `Parcelable` capabilities to a Java class in an Android app.

One approach is used by Parceler. Here, you just add a `@Parcel` annotation to the Java class, and it code generates what is needed. However, it does *not* actually make the Java class `Parcelable`. Rather, it creates a runtime wrapper class that *is* `Parcelable` and that knows how to convert instances of your own Java class to and from the wrapper. You wind up calling static `wrap()` and `unwrap()` methods on a `Parcels` class to handle the conversion between your class and the generated `Parcelable` class.

AutoParcel takes a slightly different approach. In this case, you need to:

- Add the `@AutoParcel` annotation to the class

- Make the class `abstract` and have it implement `Parcelable`
- Write `abstract` method signatures for getters for the data members

AutoParcel then code-generates a Java class that implements the getters, data members, and `Parcelable` logic, along with other niceties like `equals()` and `hashCode()`. That Java class will be named `AutoParcel_`, followed by the name of the class with the `@AutoParcel` annotation (e.g., annotating a `Foo` class gives you an `AutoParcel_Foo` class). The AutoParcel-generated class is a concrete subclass of the abstract base class, and so you can just work with the abstract class' public API and let AutoParcel handle the details.

However, neither of these give you classes that play well with the other children. Other code that expects to work with your classes — whether that is passing a Parceler-defined `Parcelable` to a third-party app or using something like Gson to handle JSON parsing — will not like either Parceler or AutoParcel that much.

By Code Generator Sites and Tools

The Parcelabler Web site is a code generator. You paste in a simple Java class, with the class declaration and data members:

```
class Book {
    String isbn;
    String title;
    int pubYear;
}
```

and it gives you an output class that adds the `Parcelable` logic:

```
class Book implements Parcelable {
    String isbn;
    String title;
    int pubYear;

    protected Book(Parcel in) {
        isbn = in.readString();
        title = in.readString();
        pubYear = in.readInt();
    }

    @Override
    public int describeContents() {
        return 0;
    }
```

```
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(isbn);
        dest.writeString(title);
        dest.writeInt(pubYear);
    }

    @SuppressWarnings("unused")
    public static final Parcelable.Creator<Book> CREATOR = new
Parcelable.Creator<Book>() {
        @Override
        public Book createFromParcel(Parcel in) {
            return new Book(in);
        }

        @Override
        public Book[] newArray(int size) {
            return new Book[size];
        }
    };
}
```

We will see in the next section what all of that code does for us, as part of understanding how to build it by hand.

However, the Parcelabler Web site has some limitations in its Java parsing, and so the more complex your Java class, the more likely it is that the Parcelabler site will have difficulty understanding it and blending in the `Parceable` logic.

The ParcelableCodeGenerator project implements a command-line code generator that takes a JSON schema and gives you a Java class that, among other things, has the `Parcelable` implementation.

By Hand

Adding `Parcelable` support yourself is not especially difficult, though it is a bit tedious.

The Parcelable Interface

The first steps is to add `implements Parcelable` to the class. Immediately, your IDE should start complaining that you need to implement two methods to satisfy the `Parcelable` interface.

The easier of the two methods is `describeContents()`, where you will return 0, most likely.

The other method you will need to implement is `writeToParcel()`. You are passed in two parameters: a very important `Parcel`, and a usually-ignored `int` named `flags`.

Your job, in `writeToParcel()`, is to call a series of `write...()` methods on the `Parcel` to write out all data members of this object that should be considered part of the object as it is passed across process boundaries. There are dozens of type-safe methods for writing data into the `Parcel`:

- methods that write individual primitives (e.g., `writeInt()`) or Java arrays of primitives (e.g., `writeStringArray()`)
- `writeBundle()`, for writing out a `Bundle`
- `writeParcelable()` and `writeParcelableArray()`, for writing out other objects that implement `Parcelable`
- `writeFileDescriptor()`, for putting a `FileDescriptor` into the `Parcel`, with an eye towards allowing whoever reconstitutes the `Parcelable` to be able to read or write a stream based on that `FileDescriptor`
- methods to write other "active objects", such as `IBinder` objects from a remote service binding
- various specialized methods for particular data types (e.g., `writeSizeF()`) or interfaces (e.g., `writeSerializable()`)

If, in `writeToParcel()`, you called `writeFileDescriptor()`, you will want to have `describeContents()` return `CONTENTS_FILE_DESCRIPTOR` instead of 0, as apparently the `Parcelable` support logic needs to know that a file descriptor is in the `Parcel`.

In the case of the generated `Book` code shown earlier in this chapter, `writeToParcel()` writes out the two `String` and one `int` data member:

```
@Override
public int describeContents() {
    return 0;
}

@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(isbn);
    dest.writeString(title);
    dest.writeInt(pubYear);
}
```

The CREATOR

When Android tries reading objects in from a `Parcel`, and it encounters an instance of your `Parcelable` class, it will retrieve a static `CREATOR` object that must be defined on that class. The `CREATOR` is an instance of `Parcelable.Creator`, using generics to tie it to the type of your class:

```
@SuppressWarnings("unused")
public static final Parcelable.Creator<Book> CREATOR = new Parcelable.Creator<Book>()
{
    @Override
    public Book createFromParcel(Parcel in) {
        return new Book(in);
    }

    @Override
    public Book[] newArray(int size) {
        return new Book[size];
    }
};
```

The `@SuppressWarnings("unused")` annotation is because the IDE will think that this `CREATOR` instance is not referred to anywhere. That is because it will only be used via Java reflection.

The `CREATOR` will need two methods. `createFromParcel()`, given a `Parcel`, needs to return an instance of your class populated from that `Parcel`. `newArray()`, given a size, needs to return a type-safe array of your class.

The typical implementation of `createFromParcel()` will delegate the actual work to a `protected` or `private` constructor on your class that takes the `Parcel` as input:

```
protected Book(Parcel in) {
    isbn = in.readString();
    title = in.readString();
    pubYear = in.readInt();
}
```

You need to read in the *same* values that you wrote out to the `Parcel`, and in the same order.

By Hand, With a Little Bit of Help

Android Studio 1.3 and higher have a template for a new `Parcelable` class. Right-click over your desired Java package and choose New > Other > New Parcelable Type from the context menu. Fill in your class and the template will create a new standalone Java class, akin to this one:

```
import android.os.Parcel;
import android.os.Parcelable;

public class Item implements Parcelable {

  // TODO declare your real class members
  // Members must be either primitives, primitive arrays or parcelables
  private int mFoo;
  private String mBar;

  // TODO implement your constructors, getters & setters, methods

  private Item(Parcel in) {
    // TODO read your class members from the parcel
    // Note: order is important - you must read in the same order
    // you write in writeToParcel!
    mFoo=in.readInt();
    mBar=in.readString();
  }

  @Override
  public void writeToParcel(Parcel out, int flags) {
    // TODO write your class members to the parcel
    // Note: order is important - you must write in the same order
    // you read in your private parcelable constructor!
    out.writeInt(mFoo);
    out.writeString(mBar);
  }

  @Override
  public int describeContents() {
    // TODO return Parcelable.CONTENTS_FILE_DESCRIPTOR if your class members
    // include a FileDescriptor, otherwise you can simply return 0
    return 0;
  }

  public static final Parcelable.Creator<Item> CREATOR=new Parcelable.Creator<Item>()
{
    public Item createFromParcel(Parcel in) {
```

```
      return new Item(in);
    }

    public Item[] newArray(int size) {
      return new Item[size];
    }
  };

}
```

You would have to adjust the stock fields (`mFoo`, `mBar`) to be what you need, and adjust the `writeToParcel()` and private constructor to match.

However, this template is designed for starting from scratch; it is not that useful when you have an existing class and wish to now make it be `Parcelable`.

The ParcelablePlease library saves you from having to do all of the reading and writing to and from the `Parcel` yourself. Putting the `@ParceablePlease` annotation on the class generates a class for you (your class name followed by `ParcelablePlease`, so `FooParcelablePlease` for a `Foo` class). This class *only* marshals your data members to and from a `Parcel`, via static `readFromParcel()` and `writeToParcel()` methods. You still have to have the rest of the `Parcelable` boilerplate. Hence, this library is not as powerful as the annotation processors mentioned earlier in this chapter, but you wind up with a "real" complete Java class that can work better with other annotation-based libraries like Gson. On the other hand, it still makes it difficult for you to distribute your code to third parties, as they will need to also have this annotation processing library in their project builds.

The Limitations of Parcelable

While the mechanics of writing a `Parcelable` are not hard, this does not mean that every model object or other POJO in your app should be made `Parcelable`. Overuse of `Parcelable` is a bit of a code smell, as it suggests that the developer is not necessarily considering all of the limitations and effects of the use of `Parcelable`.

The 1MB Limit

The biggest one (pun lightly intended) is the size limitation. A `Parcel` – the IPC structure that is used to pass `Parcelable` objects across process boundaries — has a 1MB size limit. If you get over this limit, you will likely crash with a "Failed Binder Transaction" message as part of the exception's stack trace.

There are two main ways you can reach this limit:

1. Have a `Parcelable` that individually is too large. A common case for this is wrapping a `Bitmap` or other large `byte` array in some `Parcelable` object.
2. Have too many `Parcelable` objects. For example, you might have performed a database query, converted the results into a collection of model objects, then tried to pass that collection to another activity via an `Intent` extra. Syntactically, this can work fine, if the collection and its model objects are all `Parcelable`. But now your risk of hitting the 1MB limit is determined by how many rows there are in the query's result set, and that can vary by user.

Large data like this need to be managed by singletons or other static data members and shared among your application components, rather than passed via `Parcelable` objects.

Pass-By-Value

Suppose we have two activities, A and B. Activity A calls `startActivity()`, identifying activity B in the `Intent`. The `Intent` also includes a custom `Parcelable` object, one that takes up 1KB of space.

Question: how much system RAM is taken up by that `Parcelable`?

Wrong Answer: 1KB.

Right Answer: At least 3KB, as there are at least three copies of the `Parcelable` data:

- One copy is the original `Parcelable` object, the one that is stored as an extra in the `Intent`
- Another copy is the one in the `Parcel` that is held by a core OS process, for handling things like configuration changes and the recent-tasks list, where that `Intent` (and its extras, including your `Parcelable`) are needed
- A third copy is the one in the `Intent` that Activity B receives

`Parcelable` is, in effect "pass-by-value", as the `Parceable` object is copied as part of getting it across the process boundary twice, once from your process to the core OS, and once from the core OS back to your process.

This means that modifications that Activity B makes to the `Parcelable` object will not be seen by Activity A, as they are working on separate copies of the object.

Similarly, changes that Activity B makes to the `Parcelable` will not affect the copy held by the core OS process and re-delivered to Activity B on a configuration change.

The safest way to help defend against mistakes related to this is to consider a `Parcelable` object to be an immutable object. Only configure it through a constructor (possibly with the assistance of some `Builder` if you want a cleaner API). Offer getters for the values in the `Parcelable`, but do not offer any setters, so once the instance is created, it cannot be changed.

Also note that these copies magnify the effects of having a large `Parcelable` object, or too many `Parcelable` objects in a `Parcel`. A 900KB `Parcel` might fit within the 1MB size limit, but it would consume at least 2.7MB if the `Parcel` is part of some IPC.

Conversely, there are cases where `Intent` objects are not passed across process boundaries, such as `LocalBroadcastManager`. In those cases, neither the 1MB limit nor the pass-by-value effect are an issue. Only if the `Intent` is "flattened" into a `Parcel`, and later converted back into an `Intent`, do these extra copies and the 1MB limit come into play.

The ClassLoader Conundrum

Sometimes, weird stuff happens, particularly when trying to read in other `Parcelable` objects that you wrote to the `Parcel`. In this case, the `Parcel` system needs to use Java reflection to find the Java class associated with the `Parcelable` objects, and sometimes it gets a bit lost.

When you use `readParcelable()` to read in the `Parcelable` objects out of the `Parcel`, you may need to supply the `ClassLoader` that you know has those `Parcelable` classes:

```
Foo(Parcel in) {
  this.someField=in.readParcelable(getClass().getClassLoader());
  this.anotherField=in.readParcelable(getClass().getClassLoader());
}
```

Here, we are using the same `ClassLoader` that has this `Foo` class.

Sharing Between Apps

`Parcelable` objects need to read and write the same values to and from the `Parcel`. This sounds simple, but it gets into some nasty issues when multiple code bases need to work with the `Parcelable`.

For example, suppose your app offers an SDK, such as a remote service. You have some custom `Parcelable` objects that you can either give to third-party clients of your app or get as input from those clients. Now, your SDK needs to ship implementations of the `Parcelable` classes; without them, clients cannot use you exposed service API.

What happens now, if you change the definition of the `Parcelable`? Bear in mind that:

- You may not be able to control when third-party developers take on some new version of your SDK
- You may not be able to control when end users update your app
- You may not be able to control when end users update third-party client apps

As a result, it is reasonably likely that your `Parcelable` implementations will be out of sync on a user's device, with your app having one implementation and a third-party app having another implementation. The results of this may not be pretty.

This is not a problem for purely internal uses of `Parcelable`, such as for holding onto data across a configuration change.

Beware the PendingIntent

Custom `Parcelable` objects are fine for use as extras in `Intent` objects used with `LocalBroadcastManager` or otherwise limited to your own process. Custom `Parcelable` objects *should* work when placed in the saved instance state `Bundle`.

The further you get from these scenarios, though, the more likely it is that you will run into cases where your custom `Parcelable` will cause problems. That is because other apps — and core OS processes — have no access to your `Parcelable` class. Any attempt to work with `Intent` extras will result in a crash in that other process, probably interrupting whatever it was that you were trying to do.

One example of this is using a custom `Parcelable` in an extra for an `Intent`, wrapped in a `PendingIntent`. The party that executes the `PendingIntent` has the ability to add extras to the `Intent` inside the `PendingIntent`. That, in turn, causes problems, as Android does not have access to your `Parcelable` class. You get a stack trace like this one:

```
E/Parcel: Class not found when unmarshalling:
com.commonsware.android.parcelable.marshall.Thingy
   java.lang.ClassNotFoundException:
com.commonsware.android.parcelable.marshall.Thingy
     at java.lang.Class.classForName(Native Method)
     at java.lang.Class.forName(Class.java:400)
     at android.os.Parcel.readParcelableCreator(Parcel.java:2507)
     at android.os.Parcel.readParcelable(Parcel.java:2461)
     at android.os.Parcel.readValue(Parcel.java:2364)
     at android.os.Parcel.readArrayMapInternal(Parcel.java:2717)
     at android.os.BaseBundle.unparcel(BaseBundle.java:269)
     at android.os.Bundle.putAll(Bundle.java:226)
     at android.content.Intent.fillIn(Intent.java:8171)
     at
com.android.server.am.PendingIntentRecord.sendInner(PendingIntentRecord.java:255)
     at
com.android.server.am.PendingIntentRecord.sendWithResult(PendingIntentRecord.java:216)
     at
com.android.server.am.ActivityManagerService.sendIntentSender(ActivityManagerService.java:7151)
     at android.app.PendingIntent.send(PendingIntent.java:836)
     at
com.android.server.AlarmManagerService$DeliveryTracker.deliverLocked(AlarmManagerService.java:2984)
     at
com.android.server.AlarmManagerService.deliverAlarmsLocked(AlarmManagerService.java:2424)
     at
com.android.server.AlarmManagerService$AlarmThread.run(AlarmManagerService.java:2543)
```

One workaround for cases like this is to still use a custom `Parcelable`, but instead of putting it directly as an `Intent` extra, use the `Parcel` system to convert it into a byte array, and store *that* as the extra. Foreign processes have no idea what the byte array is for and will not try to convert it into anything. When you get the byte array, you can then use the `Parcel` system to get your `Parcelable` back.

The `Parcelable/Marshall` sample project demonstrates this technique. It is a clone of the `EventBus/GreenRobot3` sample app, discussed in the chapter on event buses. The app uses `AlarmManager` to get control every minute, posting an event on a greenrobot EventBus. That event adds a row in a `ListView` if the UI is in the foreground; otherwise, the `WakefulIntentService` triggered by the alarm event will show a `Notification`.

This sample app does not really need a custom `Parcelable`. However, lots of programs have lots of things that they do not really need. So, the `Parcelable/Marshall` project adds a custom `Parcelable` class, named `Thingy`:

```
package com.commonsware.android.parcelable.marshall;

import android.os.Parcel;
import android.os.Parcelable;

public class Thingy implements Parcelable {
  final String something;
  final int anotherThing;

  public Thingy(String something, int anotherThing) {
    this.something=something;
    this.anotherThing=anotherThing;
  }

  protected Thingy(Parcel in) {
    something=in.readString();
    anotherThing=in.readInt();
  }

  @Override
  public int describeContents() {
    return(0);
  }

  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(something);
    dest.writeInt(anotherThing);
  }

  @SuppressWarnings("unused")
  public static final Parcelable.Creator<Thingy> CREATOR=
    new Parcelable.Creator<Thingy>() {
    @Override
    public Thingy createFromParcel(Parcel in) {
      return(new Thingy(in));
    }

    @Override
    public Thingy[] newArray(int size) {
      return(new Thingy[size]);
    }
```

```
  };
}
```

(from Parcelable/Marshall/app/src/main/java/com/commonsware/android/parcelable/marshall/Thingy.java)

It is fairly vanilla `Parcelable` class, wrapped around a string and an integer.

In theory, we could put a `Thingy` into the `Intent` used with `AlarmManager` via a `PendingIntent`. However, that will run into the problem outlined in this section, as Android does not have a `Thingy`. In fact, the stack trace shown above comes from this sample project, if you try putting a `Thingy` into the `Intent`.

The revised version of the project instead puts a byte array in the `Intent` as an extra, by way of the `Parcelables` utility class:

```
package com.commonsware.android.parcelable.marshall;

import android.os.Parcel;
import android.os.Parcelable;

// inspired by http://stackoverflow.com/a/18000094/115145

public class Parcelables {
  public static byte[] toByteArray(Parcelable parcelable) {
    Parcel parcel=Parcel.obtain();

    parcelable.writeToParcel(parcel, 0);

    byte[] result=parcel.marshall();

    parcel.recycle();

    return(result);
  }

  public static <T> T toParcelable(byte[] bytes,
                                   Parcelable.Creator<T> creator) {
    Parcel parcel=Parcel.obtain();

    parcel.unmarshall(bytes, 0, bytes.length);
    parcel.setDataPosition(0);

    T result=creator.createFromParcel(parcel);

    parcel.recycle();
```

```
    return(result);
  }
}
```

(from Parcelable/Marshall/app/src/main/java/com/commonsware/android/parcelable/marshall/Parcelables.java)

`Parcelables` mirrors standard Java utility classes like `Arrays` (`static` utility methods for Java arrays) and `Collections` (`static` utility methods for subclasses of `Collection`). `Parcelables` has two `static` utility methods for working with `Parcelable` objects:

- `toByteArray()` converts the `Parcelable` to a byte array
- `toParcelable()` converts the byte array back into a `Parcelable`

Both work by way of a `Parcel` object. You can get one of these from an instance pool by calling the static `obtain()` method on `Parcel`.

`toByteArray()` gets a `Parcel`, uses `writeToParcel()` to put your `Parcelable` into the `Parcel`, then uses `marshall()` to get a byte array representation of the `Parcel` contents. Before returning that result, though, we `recycle()` the `Parcel`, returning it to the instance pool for later use.

`toParcelable()` needs not only the byte array representing your object, but also your `Parcelable.Creator`, which knows how to convert a `Parcel` back into your `Parcelable`. So, `toParcelable()`:

- gets a `Parcel` via `obtain()`
- calls `unmarshall()` to populate the `Parcel` with the byte array contents
- calls `setDataPosition(0)` to effectively "rewind" the `Parcel` back to the beginning
- calls `createFromParcel()` on your `Parcelable.Creator` to get the `Parcelable` out of the `Parcel`
- recycles the `Parcel`
- and returns your `Parcelable`

The `scheduleAlarms()` method on `PollReceiver` is responsible for creating the `Intent` to schedule some alarm events. It adds a `Thingy` to the `Intent`, but uses `toByteArray()` to add the extra, rather than putting the raw `Thingy` in as the extra:

```
  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Thingy thingy=
```

```
    new Thingy(mgr.getClass().getCanonicalName(), mgr.hashCode());
  Intent i=
    new Intent(ctxt, PollReceiver.class)
    .putExtra(EXTRA_THINGY, Parcelables.toByteArray(thingy));
  PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

  mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                   SystemClock.elapsedRealtime() + INITIAL_DELAY,
                   PERIOD, pi);
}
```

(from Parcelable/Marshall/app/src/main/java/com/commonsware/android/parcelable/marshall/PollReceiver.java)

When the alarm event occurs, `PollReceiver` can get the `Thingy` back by retrieving the byte array extra and using `toParcelable()`:

```
@Override
public void onReceive(Context ctxt, Intent i) {
  Thingy thingy=
    Parcelables.toParcelable(i.getByteArrayExtra(EXTRA_THINGY),
      Thingy.CREATOR);

  if (i.getAction() == null) {
    ScheduledService.enqueueWork(ctxt);
  }
  else {
    scheduleAlarms(ctxt);
  }
}
```

(from Parcelable/Marshall/app/src/main/java/com/commonsware/android/parcelable/marshall/PollReceiver.java)

While this approach will add a few lines of code to your project, it should not incur significant additional overhead. All the work that is being done here is part and, um, parcel of passing a `Parcelable` between processes anyway. We are just doing it proactively, to eliminate any references in the `Parcel` to our custom `Parcelable` class.

Responding to URLs

You may have noticed that Android supports a `market:` URL scheme. Web pages can use such URLs so that, if they are viewed on an Android device's browser, the user can be transported to a Play Store page, perhaps for a specific app or a list of apps for a publisher.

Fortunately, that mechanism is not limited to Android's code — you can get control for various other types of links as well. You do this by adding certain entries to an activity's `<intent-filter>` for an `ACTION_VIEW Intent`.

However, be forewarned that this capability is browser-specific. What works on the original Android "Browser" app and Google's Chrome may not necessarily work on Firefox for Android or other browsers.

Prerequisites

Understanding this chapter requires that you have read the chapter on `Intent` filters.

Manifest Modifications

First, any `<intent-filter>` designed to respond to browser links will need to have a `<category>` element with a name of `android.intent.category.BROWSABLE`. Just as the `LAUNCHER` category indicates an activity that should get an icon in the launcher, the `BROWSABLE` category indicates an activity that wishes to respond to browser links.

You will then need to further refine which links you wish to respond to, via a `<data>` element. This lets you describe the URL and/or MIME type that you wish to respond

to. For example, here is the `AndroidManifest.xml` file from the Introspection/URLHandler sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.commonsware.android.urlhandler" android:versionCode="1" android:versionName="1.0">

  <uses-sdk android:minSdkVersion="14" android:targetSdkVersion="19"/>

  <supports-screens android:largeScreens="true" android:normalScreens="true" android:smallScreens="false"/>

  <application android:icon="@drawable/ic_launcher" android:label="@string/app_name">
    <activity android:name="URLHandler" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>

        <data android:mimeType="application/pdf"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>

        <data android:host="www.this-so-does-not-exist.com" android:path="/something"
android:scheme="http"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.android.MY_ACTION"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Introspection/URLHandler/app/src/main/AndroidManifest.xml)

Here, we have four `<intent-filter>` elements for our one activity:

- The first is a standard "put an icon for me in the launcher, please" filter, with the `LAUNCHABLE` category
- The second claims that we handle PDF files (MIME type of `application/pdf`), and that we will respond to browser links (`BROWSABLE` category)

- The third claims that we will handle any HTTP request (scheme of `"http"`) for a certain Web site (host of `"www.this-so-does-not-exist.com"` and path of `/something`), and that we will respond to browser links (`BROWSABLE` category)
- The last is a custom action, for which we will generate a URL that Android will honor, and that we will respond to browser links (`BROWSABLE` category) — we will examine this more closely in the next section

What happens for the first two links varies based on browser.

The original Android "Browser" app, and Google Chrome, will do the following:

- Tapping the link to the PDF, on Android 2.3+, will trigger a download of the PDF. When the user taps on the downloaded file (e.g., from the `Notification` in the status bar), the user will have `URLHandler` as one of the options in the chooser to view the PDF file.
- Tapping the link to `http://www.this-so-does-not-exist.com/something` will bring up a chooser showing all available Web browser, plus `URLHandler`, as expected

Firefox for Android will treat the PDF link the same way. However, Firefox for Android does not check the URL for the second link to see if there is anything else supporting `ACTION_VIEW` for the URL, and so it always loads up the Web page. You see this effect with the link to Barcode Scanner as well — even though a device has Barcode Scanner installed, Firefox never offers that as an option.

Creating a Custom URL

Responding to MIME types makes complete sense... if we implement something designed to handle such a MIME type.

Responding to certain schemes, hosts, paths, or file extensions is certainly usable, but other than perhaps the file extension approach, it makes your application a bit fragile. If the site changes domain names (even a sub-domain) or reorganizes its site with different URL structures, your code will break.

If the goal is simply for you to be able to trigger your own application from your own Web pages, though, the safest approach is to use an `intent:` URL. These can be generated from an `Intent` object by calling `toUri(Intent.URI_INTENT_SCHEME)` on a properly-configured `Intent`, then calling `toString()` on the resulting `Uri`.

For example, the `intent:` URL for the fourth `<intent-filter>` from above is:

```
intent:#Intent;action=com.commonsware.android.MY_ACTION;end
```

This is not an official URL scheme, any more than `market:` is, but it works for Android devices. When the Android built-in Browser encounters this URL, it will create an `Intent` out of the URL-serialized form and call `startActivity()` on it, thereby starting your activity. Chrome also supports this URL structure. Firefox for Android does not, indicating instead that it cannot recognize the URL.

Reacting to the Link

Your activity can then examine the `Intent` that launched it to determine what to do. In particular, you will probably be interested in the `Uri` corresponding to the link — this is available via the `getData()` method. For example, here is the `URLHandler` activity for this sample project:

```
package com.commonsware.android.urlhandler;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class URLHandler extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TextView uri=(TextView)findViewById(R.id.uri);

    if (Intent.ACTION_MAIN.equals(getIntent().getAction())) {
      String intentUri=(new Intent("com.commonsware.android.MY_ACTION"))
                          .toUri(Intent.URI_INTENT_SCHEME)
                          .toString();

      uri.setText(intentUri);
      Log.w("URLHandler", intentUri);
    }
    else {
```

```
      Uri data=getIntent().getData();

      if (data==null) {
        uri.setText("Got com.commonsware.android.MY_ACTION Intent");
      }
      else {
        uri.setText(getIntent().getData().toString());
      }
    }
  }

  public void visitSample(View v) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse("https://commonsware.com/sample")));
  }
}
```

(from Introspection/URLHandler/app/src/main/java/com/commonsware/android/urlhandler/URLHandler.java)

This activity's layout has a `TextView` (uri) for showing a `Uri` and a `Button` to launch a page of links, found on the CommonsWare site (`https://commonsware.com/sample`). The `Button` is wired to call `visitSample()`, which just calls `startActivity()` using the aforementioned URL to display it in the user's chosen Web browser.

When the activity starts up, though, it first loads up the `TextView`. What goes in there depends on how the activity was launched:

1. If it was launched via the launcher (e.g., the action is `MAIN`), then we display in the `TextView` the `intent:` URL shown in the previous section, generated from an `Intent` object designed to trigger our fourth `<intent-filter>`. This also gets dumped to Logcat, and is how the author got this URL in the first place to put on the sample Web page of links.
2. If it was not launched via the launcher, it was launched from a Web link. If the `Uri` from the launching `Intent` is `null`, though, that means the activity was launched via the custom `intent:` URL (which only has an action string), so we put a message in the `TextView` to match.
3. Otherwise, the `Uri` from the launching `Intent` will have something we can use to process the link request. For the PDF file, it will be the local path to the downloaded PDF, so we can open it. For the `www.this-so-does-not-exist.com` URL, it will be the URL itself, so we can process it our own way.

Note that for the PDF case, clicking the PDF link in the Browser will download the file in the background, with a `Notification` indicating when it is complete. Tapping on the entry in the notification drawer will then trigger the `URLHandler` activity.

Also, bear in mind that the device may have multiple handlers for some URLs. For example, a device with a real PDF viewer will give the user a choice of whether to launch the downloaded PDF in the real view or `URLHandler`.

App Links

We have had the ability to have activities with `<intent-filter>` elements that support custom schemes (e.g., `myapp://`) since Android 1.0. The benefit over using a custom scheme is that, if it is unique on the device, an `Intent` for that custom scheme will go straight to the desired activity. However, this approach had a lot of flaws:

- There is no guarantee of uniqueness
- Few apps would recognize the custom scheme and issue an `ACTION_VIEW` `Intent` on the desired `Uri`
- If the user *did* encounter a link that would try to issue the `ACTION_VIEW` `Intent`, and the app handling that custom scheme was not installed, the request would simply fail

Using an `<intent-filter>` advertising support for some `http` or `https` URL would improve the results for the latter two issues, as many more apps would recognize the URL as being a URL, and usually the fallback would be to have a browser open up on that URL. However, now it is guaranteed that the scheme is *not* unique. Users would initially get a chooser, to determine what activity should handle the request. This can be confusing, particularly since the chooser does not really indicate the scope of the choice (would I be saying that XYZ app is now handling all Web links?).

Android 6.0 added an interesting solution for this. If you use a `<intent-filter>` for a domain that you control, you can publish a bit of metadata, as a JSON file, on the Web server. Android can be taught to sniff for that metadata and use it to validate that the app was developed by the same person or group that runs the server for the identified domain. In that case, Android will bypass the chooser and go straight to the activity with the domain-specific `<intent-filter>`. The cited example would be Twitter doing this, so any link click on a `twitter.com` URL would bring up the Twitter app, not a Web browser.

Of course, these links are only so useful. They are fine for when a link appears in an ordinary app. Web browsers, however, tend not to actually see whether a URL they encounter is handled by some on-device app. Android 6.0 does not change this behavior. So, links on Web pages viewed in 2015 versions of Firefox will not honor your desired `<intent-filter>` regardless of whether you are using this new app link system or not. Chrome's behavior varies by version.

That being said, app links still have their uses (e.g., responding to links from social media posts).

Setting Up the IntentFilter

Supporting an `<intent-filter>` for some `http` or `https` URL has been possible since Android 1.0. The only thing that is different is that now you can add an `android:autoVerify="true"` to the `<intent-filter>` element, to tell Android that you would like it to verify the connection between the app and the domain used in the `<intent-filter>`, to skip the chooser when URLs for that domain trigger your `<intent-filter>`.

For example, the Introspection/URLHandlerMNC sample project is a revised version of the `URLHandler` sample, one that switches its `http` `<intent-filter>` to look for `https://commonsware.com` URLs, and it incorporates `android:autoVerify="true"`:

```
    <intent-filter android:autoVerify="true">
      <action android:name="android.intent.action.VIEW" />

      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />

      <data
        android:host="commonsware.com"
        android:scheme="https" />
    </intent-filter>
```

(from Introspection/URLHandlerMNC/app/src/main/AndroidManifest.xml)

On pre-Marshmallow versions of Android, this attribute will be ignored, as it will not be recognized. But, on Android 6.0+, this attribute will be used to attempt to validate that your app was written by somebody who owns the specified domain.

The author of this book owns the `commonsware.com` domain. To actually run this project and have the updated app linking work, you would need to switch this to be some domain that *you* control.

Note that while `android:autoVerify="true"` is written at the scope of a single `<intent-filter>`, it affects *all* activities and *all* `<intent-filter>` structures. *All* of them that use `http` or `https` as the `android:scheme` must support the app links protocol described in this chapter. You cannot have some filters supporting app links and others not — either they *all* support app links, or none will.

Setting Up the JSON

When Android installs an app that has one or more `<intent-filter>` elements with `android:autoVerify="true"`, it will attempt to find a JSON file on the identified server. Specifically, for the sample app, Android will create a URL of the form:

```
https://commonsware.com/.well-known/assetlinks.json
```

In your app, `commonsware.com` would be replaced with the domain you have in your `<intent-filter>`.

This URL is part of a proposed IETF standard that unfortunately does not appear to be formally documented.

Android 6.0+ will use HTTPS to retrieve your `assetlinks.json` file, regardless of the scheme that you use in the `<intent-filter>`. Also, the JSON needs to be publicly accessible, without any forms of authentication. And, the JSON needs to be served with a MIME type of `application/json`.

The JSON content itself is an array of JSON objects, one object per application ID that you publish as an app:

```
[
  {
    "relation": ["delegate_permission/common.handle_all_urls"],
    "target": {
      "namespace": "android_app",
      "package_name": "com.commonsware.android.urlhandler",
      "sha256_cert_fingerprints": ["A9:99:84:D8:...:60:5B:CB:E3"]
    }
  }
]
```

(the `sha256_cert_fingerprints` value is shown truncated for easier reading)

Here, the only two variable bits are:

1. The `package_name`, which will be your application ID, and
2. The `sha256_cert_fingerprints` array, which will list the SHA256 hashes of your public signing keys, for whatever keystores you might be using for this app (e.g., your debug keystore and your production keystore)

To get the SHA256 hash of your public signing key, you will need to use the `keytool` command from your Java SDK (Java 7 or higher required):

```
keytool -list -v -keystore ...
```

where `...` is the path to your keystore (e.g., `~/.android/debug.keystore` for your debug keystore on macOS and Linux).

You will need to provide the password to the keystore. For the debug keystore, this is `android`.

As part of the output, you will get the SHA256 hash:

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: androiddebugkey
Creation date: Aug 7, 2011
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4e3f2684
Valid from: Sun Aug 07 19:57:56 EDT 2011 until: Tue Jul 30 19:57:56 EDT 2041
Certificate fingerprints:
   MD5:  98:84:0E:36:F0:B3:48:9C:CD:13:EB:C6:D8:7F:F3:B1
   SHA1: E6:C5:81:EB:8A:F4:35:B0:04:84:3E:6E:C3:88:BD:B2:66:52:E7:09
   SHA256: A9:99:84:D8:...:60:5B:CB:E3
   Signature algorithm name: SHA1withRSA
   Version: 3


*******************************************
*******************************************
```

(the SHA256 value is shown truncated for easier reading)

That long set of hex digits will need to go in the `sha256_cert_fingerprints` JSON array.

The rest of the JSON is fixed. Try not to introduce other JSON properties and such into this file, as they may cause your file to fail validation. However, you can have multiple JSON objects for multiple apps, each providing the `relation` and `target` properties.

Results

Our `URLHandler` activity not only responds to `http://misc.commonsware.com` URLs, but it uses one if the user taps the "view-sample" button:

```
package com.commonsware.android.urlhandler;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class URLHandler extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (Intent.ACTION_VIEW.equals(getIntent().getAction())) {
      findViewById(R.id.visit).setEnabled(false);
    }
  }

  public void visitSample(View v) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse("https://commonsware.com/Android/")));
  }
}
```

(from Introspection/URLHandlerMNC/app/src/main/java/com/commonsware/android/urlhandler/URLHandler.java)

There, we launch an `ACTION_VIEW` `Intent` on a `http://commonsware.com/Android` URL via `startActivity()`.

On a pre-Marshmallow device, this `startActivity()` request will normally bring up a chooser, offering the `URLHandler` activity along with Web browsers and potentially other apps.

On an Android 6.0+ device, in the normal case, if the server is configured properly with the above JSON, and if the app was compiled by the author of this book, the chooser is bypassed, and the user gets another instance of `URLHandler`. The "another instance" part can be controlled via `Intent` flags or manifest entries, as is covered in the chapter on tasks.

However, this is not assured:

- If *you* compile and run the app, your signing key should not match the JSON-published fingerprint, and so the validation will fail and normal chooser behavior will return. You would have to substitute some URL of your own with a corresponding JSON file on that server that contains your hash.
- If the server is mis-configured (e.g., JSON not available via HTTPS), the validation will fail and normal chooser behavior will return.
- If the app is not signed with the correct signing key — such as the user is really running a copy of your app with injected malware and somebody else's signing key — the validation will fail and normal chooser behavior will return.
- If there is no connectivity at the time the user installs the app (e.g., they are side-loading it), the validation will fail and normal chooser behavior will return. The device may try to validate again in the future, though.

User Intervention

Another thing that can change the behavior to return is if the user revokes the app link. Users can do this by going to the app's screen in the Settings app and clicking the "Open by default" option:

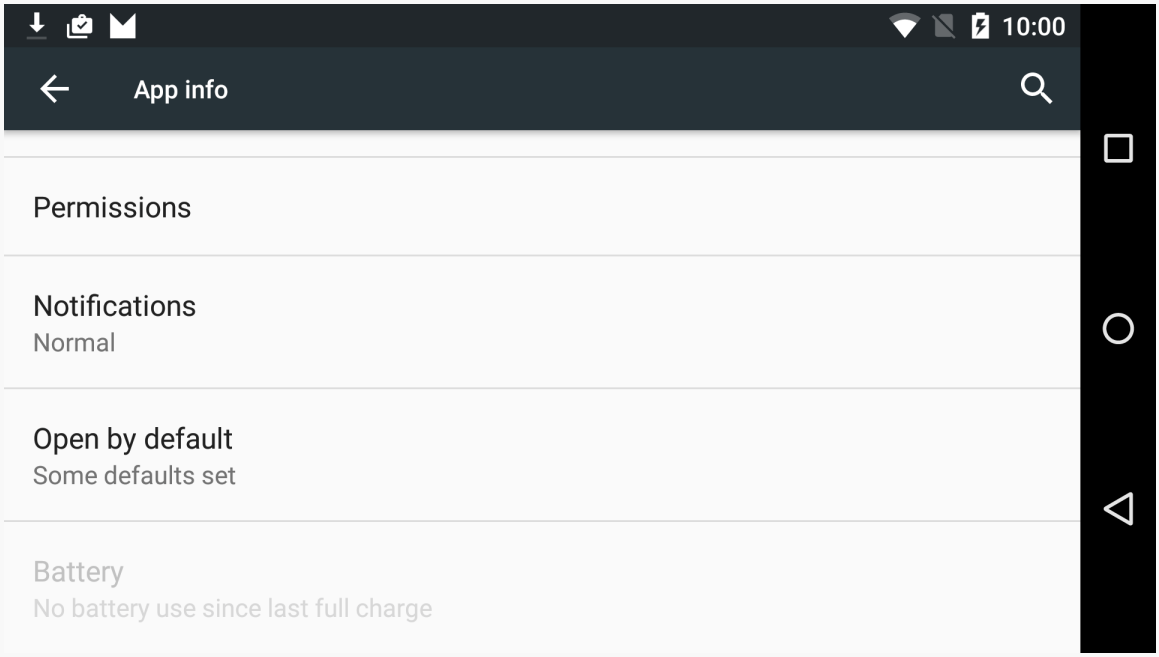


Figure 888: URLHandlerMNC in Settings, "Open by default" Visible

If the user taps that entry, one section of the next screen is entitled "App links" and gives the user the option to toggle the app link behavior off:

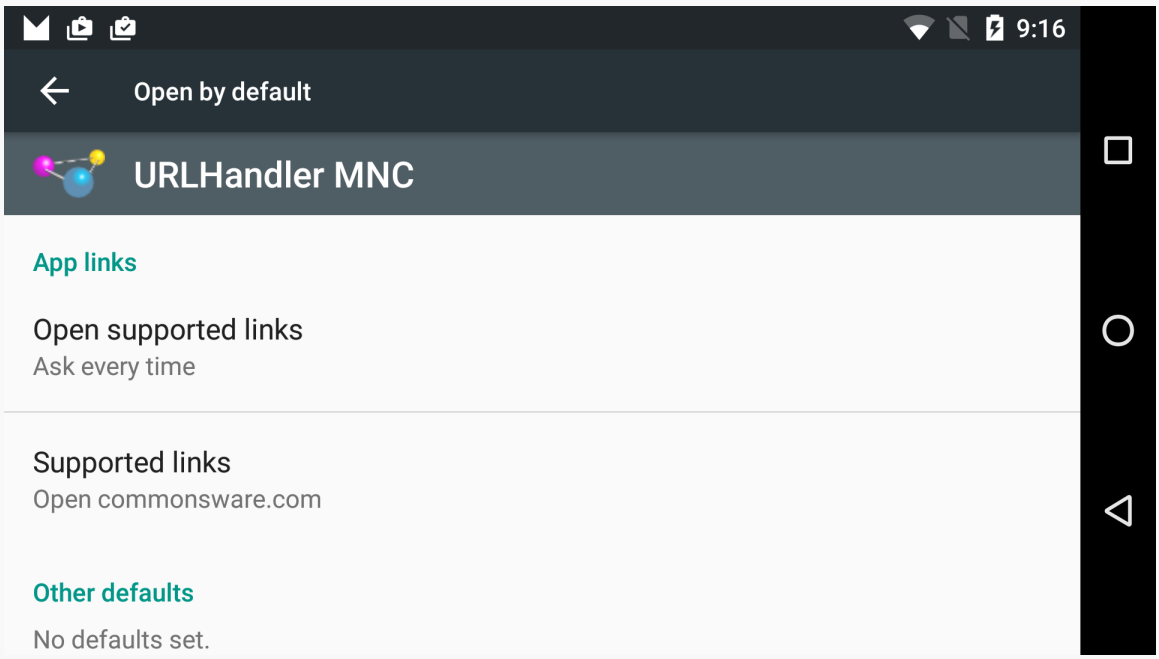


Figure 889: URLHandlerMNC in Settings, "Open by default" Screen

Unfortunately, the labeling here does not seem to work properly. The "Ask every time" choice shown selected here actually bypasses the chooser. The available choices are "open in this app", "ask every time", and "don't open in this app":

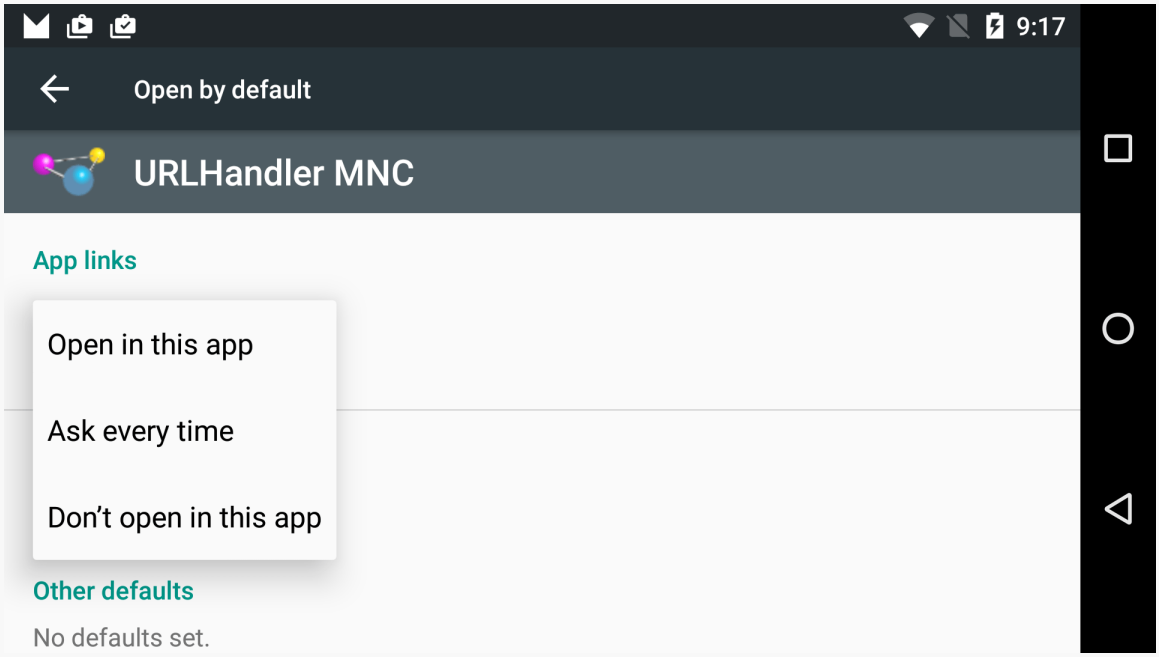


Figure 890: URLHandlerMNC in Settings, "Open supported links" Options

Testing Your Setup

You can confirm that other parties can see your `assetlinks.json` file by visiting the following URL:

```
https://digitalassetlinks.googleapis.com/v1/statements:list?
  source.web.site=https://DDDDD&
  relation=delegate_permission/common.handle_all_urls
```

(NOTE: the URL shown above is split across several lines for readability but should be all on one line when actually using the URL)

Replace `DDDDD` with the domain name for your site, and you should get a JSON document back that, among other things, contains the details from your `assetlinks.json` file:

```
{
  "statements": [
    {
      "source": {
        "web": {
          "site": "https://commonsware.com."
        }
```

```
    },
    "relation": "delegate_permission/common.handle_all_urls",
    "target": {
      "androidApp": {
        "packageName": "com.commonsware.android.urlhandler",
        "certificate": {
          "sha256Fingerprint": "A9:99:84:D8:...:60:5B:CB:E3"
        }
      }
    }
  },
  "maxAge": "3213.779933024s"
}
```

(`sha256Fingerprint` truncated for readability)

If you try visiting that URL, and there is no `assetlinks.json` file available for that domain, you will get a JSON response back containing a `debugString` indicating the nature of the problem.

You can see if an Android device in your lab has successfully performed the app link validation by running the **`adb shell dumpsys package domain-preferred-apps`** command. This will list all of the apps that have app links, and your app should appear among them, in a stanza like this one:

```
  Package: com.commonsware.android.urlhandler
  Domains: commonsware.com
  Status:  never
```

The status will reflect the user's choice of how to handle your app link inside of Settings (the `never` shown above indicates that the user decided to ignore your app link and have your app never handle such URLs).

App Shortcuts

Your app probably has a single activity that appears in the user's home screen launcher. It is the activity that has the `<intent-filter>` for the `MAIN` action and the `LAUNCHER` category.

For years, many home screens for Android have allowed the user to make "shortcuts" to that activity, typically by long-pressing the icon in the launcher, then dragging it to the desired spot on the home screen. This is reminiscent of similar capabilities in many desktop operating systems.

However, some desktops have gone beyond that. For example, with the Unity desktop in Linux, right-clicking a launcher icon in the Unity dock may bring up specific ways to get into the app identified by that icon. For example, an email client might offer "Compose New Message" from the icon's context menu, so whereas a simple click on the icon would bring up the inbox, right-clicking and choosing "Compose New Message" would bring up a message composer.

Android 7.1 adds the awkwardly-named "app shortcuts" to mimic this sort of feature. There are two ways of adding these shortcuts: via a resource tied into the manifest, and via Java code. The former approach has no particular ties to Android 7.1, and third-party home screen implementations are already adopting it.

In this chapter, we will explore what app shortcuts are, how to add them to the manifest, and how to offer "dynamic" app shortcuts from Java.

Prerequisites

Understanding this chapter requires that you have read the chapter on `Intent` filters.

Enabling Deep Dives

Google has been steadily increasing the ways in which users can drive directly into specific portions of your app, as opposed to always getting into it via a home screen launcher or perhaps the overview screen, such as:

- Notifications allow you to offer several actions in addition to the "main" action of tapping on your tile in the notification shade, to let the user go directly to where they want to go (or take action straight from the notification, bypassing your activity-based UI)
- App links allow specific URLs to drive directly into whatever portion of your app makes sense, perhaps even bypassing the normal chooser
- Direct share targets allow you to drive the user to some specific portion of your app when they elect to "share" some content via `ACTION_SEND`

All of these are designed to make it a bit easier for power users to get where they want to go quickly, saving some taps, swipes, or other forms of input.

The app shortcuts added by Android 7.1 work much the same way.

App Shortcuts, from the User's POV

It will help to understand what you are supposed to be adding to your app if you see what the user experience is for apps with app shortcuts.

However, technically, a home screen can do whatever it wants with app shortcuts, from a presentation standpoint. So, let's focus on the Pixel Launcher first, which is the launcher that Google shipped with their 2016 Pixel phones and offers app shortcut support. The Android 7.1 emulator has a similar launcher.

Ad-Hoc Requests

The user can long-press on an app icon and pull up a list of available app shortcuts:

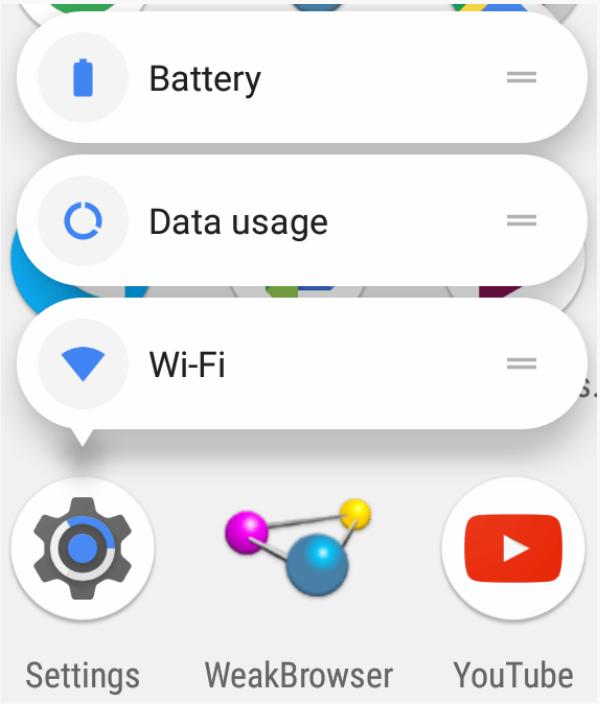


Figure 891: App Shortcuts for Settings App

If the icon does not support app shortcuts, the long-press simply does whatever it ordinarily would have done prior to app shortcuts. For example, long-pressing an icon in the launcher would allow the user to drag it to be a shortcut on the main home screen. To do those sorts of things with an icon that *does* support app shortcuts, you not only need to long-press but also start dragging the icon somewhere.

Pinning

Each of those app shortcuts has a small "grab handle" (looks like =). The user can drag that and use it to create a shortcut on the home screen for that particular app shortcut:

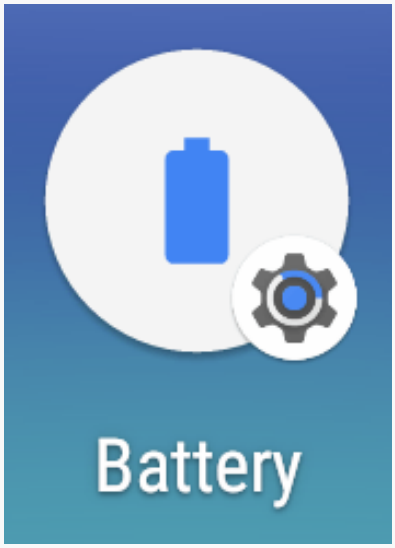


Figure 892: Pinned Battery App Shortcut from Settings App

Tapping that icon directly launches whatever the app shortcut has specified.

Alternatives

However, developers are limited only by their imaginations in terms of presentation of app shortcuts. Home screens have easy access to app shortcut information via the `LauncherApps` utility class, and there is little stopping other apps from doing the same. So, you can imagine:

- An app widget that makes app shortcuts available for a particular app, without having to manually pin them
- A launcher for a mouse-centric device offering a floating panel of app shortcuts when the user hovers a mouse over a launcher icon
- A launcher for a keyboard-centric device offering users the ability to "pin" app shortcuts to key combinations
- An app that converts an app shortcut into a notification shade tile

And so on.

Offering Manifest App Shortcuts

The "low-hanging fruit" of app shortcuts is to offer some static options via the manifest. This takes very little time to implement, including no mandatory Java code

changes. Furthermore, while the Android 7.1 APIs for working with app shortcuts may not exist on older devices, home screens and other apps could still support manifest app shortcuts with a bit of additional code. Hence, the app shortcuts that you offer via the manifest will become available to the users of many popular alternative home screen implementations, in addition to users of Android 7.1+ devices.

The `AppShortcuts/WeakBrowser` sample project demonstrates the use of both manifest and dynamic app shortcuts. This app implements a silly little Web browser, allowing the user to visit a handful of hard-coded sites.

Identify the Destinations

First, you need to decide where these app shortcuts should send users.

From a navigation flow standpoint, an app shortcut:

- Should improve efficiency of power users, saving them clicks elsewhere
- Should *not* be uniquely accessible via app shortcuts, for users who lack them or do not know about them

For example, suppose that a common bit of existing navigation in your app is:

- User taps on launcher icon
- User taps on a tab in the activity, which leads them to a particular destination

Adding an app shortcut to that same destination is easy but not that useful, as it will be no faster — and perhaps slower — to activate the app shortcut than it would to be to just go into the activity and tap on the desired tab.

But, the navigation might be more complex, where getting to the destination:

- Requires opening a navigation drawer
- Requires scrolling through a list or grid, which might be lengthy
- Requires executing some sort of search

Now offering rapid access to the destination via an app shortcut may be useful, as it may be faster than the ordinary navigation options.

Of course, `WeakBrowser`, being weak, has one manifest app shortcut: to allow the user to visit a search engine. This same page is available by tapping a "search" action bar item. This is not an especially effective use of manifest app shortcuts, but it helps to simplify the example.

Ensure the Destination is "Evergreen"

App shortcuts offered via the manifest are static. You cannot modify them at runtime, the way that you can with dynamic app shortcuts. Hence, they cannot really be personalized.

Also, if the user pins one of these app shortcuts, and some future version of your app eliminates the app shortcut, the pinned app shortcut may remain on the user's home screen. Tapping it would display some sort of "you cannot do this anymore" message. From a user experience standpoint, this will not be popular.

So, try to have your manifest app shortcuts be "evergreen", ones that are unlikely to need to be changed or removed in the future.

Add Entry Points for Destination in Manifest

An app shortcut triggers a call to `startActivity()` on some `Intent`. With manifest app shortcuts, you describe the `Intent` in XML, and some other process creates that `Intent` and passes it to `startActivity()`.

This means that any destination that you want to offer needs to be able to be reached by some `startActivity()` call, with an `Intent` that can be built out of some combination of the following:

- action string
- a `Uri` (the `data` facet of an `Intent`)
- a MIME type
- a target class and package name of the desired activity

Notably, it appears that you cannot use extras or categories to distinguish this `Intent` from any other that starts up the same activity.

Also, this activity will need to be exported, as third-party apps will need to be able to start up the activity. If the activity has an `<intent-filter>`, it will be exported. Otherwise, you will need to add `android:exported="true"` to the `<activity>` in the manifest.

Write the XML

The manifest app shortcuts are defined via an XML resource, usually residing in `res/xml/` within your module's `main/` source set. This will contain a root `<shortcuts>` element, which itself contains one or more `<shortcut>` elements:

```
<?xml version="1.0" encoding="utf-8"?>
<shortcuts xmlns:android="http://schemas.android.com/apk/res/android">
  <shortcut
    android:icon="@drawable/ic_search_black_24dp"
    android:shortcutId="search"
    android:shortcutLongLabel="@string/search_long_desc"
    android:shortcutShortLabel="@string/search">
    <intent
      android:action="android.intent.action.SEARCH"
      android:targetClass="com.commonsware.android.appshortcuts.MainActivity"
      android:targetPackage="com.commonsware.android.appshortcuts" />
  </shortcut>
</shortcuts>
```

(from AppShortcuts/WeakBrowser/app/src/main/res/xml/shortcuts.xml)

Here, we have a single app shortcut. The required attributes are:

- `android:shortcutId`, for a unique identifier for this app shortcut
- `android:icon`, for a launcher-style icon for this app shortcut
- `androdid:shortcutShortLabel`, which will be the caption for the app shortcut icon

`android:shortcutLongLabel` is optional. In theory, it will be used in places where a longer description of the app shortcut may be useful. In practice, it is unclear where this would be used.

The other required piece of a shortcut definition is the nested `<intent>` element. This describes what `Intent` should be used with `startActivity()` to take the user to where this app shortcut advertises as its destination. Typically, you will use the three attributes shown in the above sample:

- `android:action`, for an action string, as this is required, even if it is totally useless
- `android:targetClass` and `android:targetPackage`, to provide the pieces of the `ComponentName` to identify the activity to be started

A shortcut can have several `<intent>` elements, which will cause Android to create a fake back stack for the user (i.e., pressing BACK from the last `<intent>` will take the user to whatever activity was identified in the preceding `<intent>`). And, a `<shortcuts>` element can have one or several `<shortcut>` elements. However, bear in mind that a launcher may not use many app shortcuts — for example, the Pixel Launcher seems to cap the presentation at three app shortcuts. These results will vary by launcher (or other app shortcuts client) but you should assume that you only have so many app shortcut "slots" to display to the user.

Add to the Manifest

Then, you need to add a `<meta-data>` element to your `<activity>` element for your launcher activity in the manifest, pointing Android to your XML resource:

```
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <meta-data
        android:name="android.app.shortcuts"
        android:resource="@xml/shortcuts" />
    </activity>
```

(from AppShortcuts/WeakBrowser/app/src/main/AndroidManifest.xml)

Only app shortcuts declared on launcher activities will be honored. If you try putting this `<meta-data>` element on other activities, it will be ignored. If your app is one of the few with multiple launcher icons, each could have its own app shortcuts. Or, you might take this opportunity to consolidate those launcher icons into a single one, with the secondary launcher icons turning into app shortcuts.

The fact that this is just a `<meta-data>` element and an XML resource is why existing home screens could adopt manifest app shortcuts. All of this information is available via `PackageManager`, going back to the earliest Android versions.

Results

If you install this app on a device with a compatible home screen implementation, the manifest app shortcut should be available, such as what you get on the Pixel:

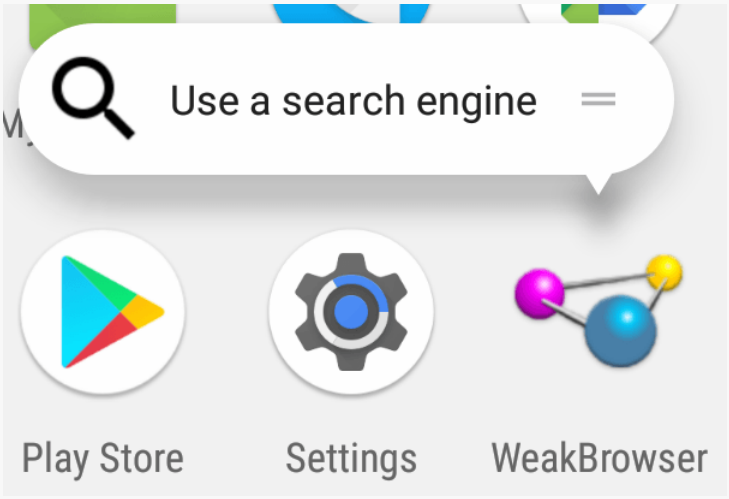


Figure 893: Manifest App Shortcut for WeakBrowser

The biggest problem comes with the icons. There are no instructions at all as what these icons should look like, or what size they should be. The author's assumption is that they should be launcher-style icons is made in part by the behavior of when you use *other* types of icons, such as the simple action bar-style search icon:

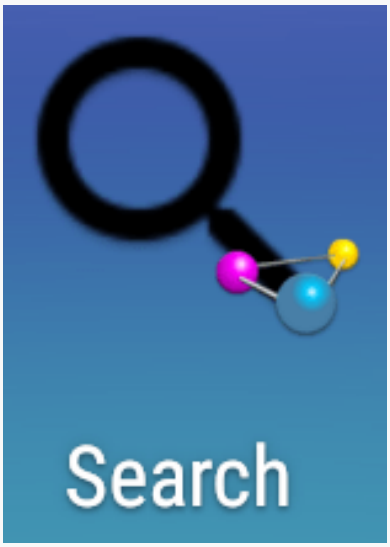


Figure 894: Pinned Manifest App Shortcut for WeakBrowser

Here, the app shortcut was pinned to the home screen, and the icon looks... unpleasant.

Also note that while the Pixel Launcher superimposes your app's icon over the app shortcut icon, it is unclear if that is something that is required by the framework or merely a Pixel Launcher convention.

Disabling Manifest App Shortcuts

So, you ship an update to your app, where you declare some manifest app shortcuts. Some time later, you revamp the UI of your app, and one of those app shortcuts no longer makes sense. It might not even work anymore — for example, you might have removed support for the activity that the app shortcut pointed to.

You might think that whatever shortcut XML you use in the new app version is what the device will use, once the user upgrades to the new app version. That is true, with one noteworthy exception: pinned app shortcuts. Google does not want these to vanish into thin air based on an app update, as that might confuse the user.

You have two main options for how to handle this gracefully:

1. Your revised shortcut XML might *repurpose* the existing app shortcut. Have a `<shortcut>` with the same `android:shortcutId` attribute, but give it whatever icon, labels, and `<intent>` are appropriate. Users who upgrade your app will have their pinned shortcut updated to reflect the new settings. This works well in cases where the app shortcut has changed a bit but still closely resembles its original role.
2. Your revised shortcut XML might *disable* the existing app shortcut. This would happen if your shortcut XML lacked any `<shortcut>` for the old ID. Preferably, though, you have a `<shortcut>` for the to-be-disabled ID. On that element, you can have `android:enabled="false"` to indicate that the app shortcut is now disabled, and you can have `android:shortcutDisabledMessage` pointing to a string resource where you explain why that app shortcut has been disabled. If the user taps on the app shortcut, this message should appear.

Offering Dynamic App Shortcuts

Truly personalized app shortcuts usually cannot be specified in the manifest. For example, you may want to allow the user to have an app shortcut to their favorite "friend" in your social network client. The identity of that friend varies by user and time. While you could offer a manifest-registered app shortcut for "Favorite Friend",

the user will not know necessarily who that friend is. With dynamic shortcuts, you can craft one that uses the name and avatar of that specific friend.

Offering dynamic app shortcuts is more powerful and correspondingly more complex.

Our `WeakBrowser` sample app, on initial install, only has the one manifest app shortcut. However, the user can visit a Settings activity within the app and elect to enable "bookmarks":

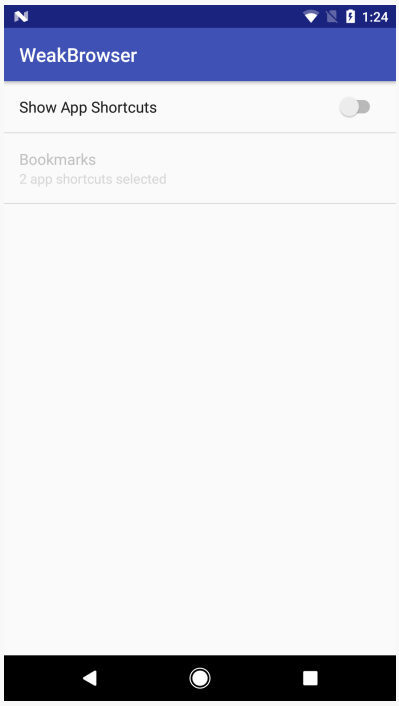


Figure 895: WeakBrowser Settings Activity

The user can choose which bookmarks to use from a multi-selection preference:

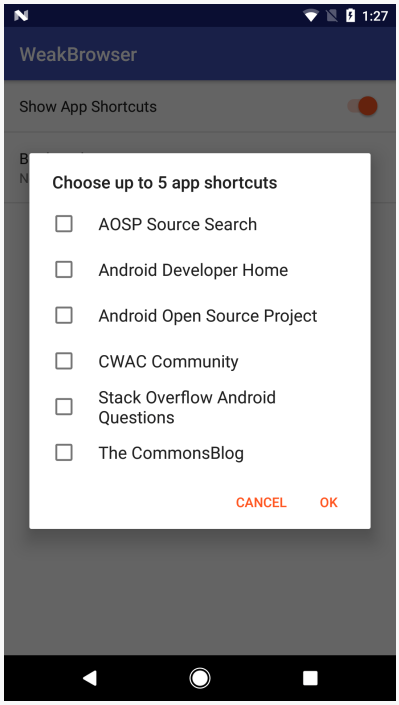


Figure 896: WeakBrowser Settings Activity, Showing Bookmarks

If the user checks some bookmarks, they get added as dynamic app shortcuts, to go along with the existing manifest app shortcut:

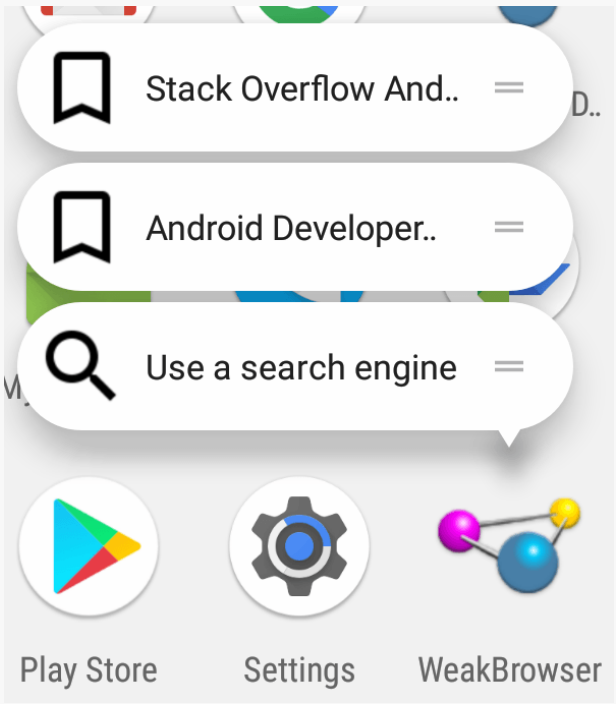


Figure 897: WeakBrowser Manifest and Dynamic App Shortcuts

Grok the Adjectives

An app shortcut is "pinned" if, by one means or another, the user has indicated that they want long-term direct access to whatever that app shortcut represents. In the Pixel Launcher, an app shortcut is pinned if the user grabs the grab handle and drags it as a shortcut onto the home screens. Different home screens will have different visual metaphors for "pinned".

An app shortcut is "immutable" if it cannot be changed by the app that provided ("published") the app shortcut. Manifest app shortcuts are immutable. Conversely, an app shortcut is "mutable" if its contents can be changed. Dynamic app shortcuts are mutable.

Ponder the IDs

Each app shortcut has a unique ID. For manifest app shortcuts, that is set via the `android:shortcutId` attribute in the `<shortcut>` element. Dynamic app shortcuts have an equivalent means of establishing their ID.

As you manipulate dynamic app shortcuts, what happens depends upon:

- What operation are you doing (setting? adding? updating? removing?)
- Whether the app shortcut ID(s) associated with the operation matches existing app shortcut ID(s) from your app
- Whether those app shortcuts are mutable or immutable
- Whether the user has pinned any of those app shortcuts

Identify the Destinations

As with manifest app shortcuts, you need to know where you are going to send the user within your app when the user chooses one of your dynamic app shortcuts. However, in this case, you will be able to provide a full `Intent` associated with the app shortcut. In principle, you could use things like extras, whereas that is not documented to be supported for manifest app shortcuts.

Craft the Intent

As with manifest app shortcuts, the destination for your dynamic app shortcuts needs to be identifiable by an `Intent` that will be used with `startActivity()` to take the user to that destination. However, unlike with manifest app shortcuts, you have full control over the setup of the `Intent` that is used for dynamic app shortcuts.

In particular, you may want to consider what `Intent` flags to use. A manifest app shortcut will have `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_CLEAR_TASK` added to the `Intent` constructed from the shortcut XML. This will send the user to your destination, wiping out the back stack from that task. You might elect to use other flags, or control things using `<activity>` manifest attributes like `android:taskAffinity`, to get the flow that you want.

For the bookmarks, our sample app has a model class, named `Bookmark`, much to nobody's surprise:

```
package com.commonsware.android.appshortcuts;

import java.util.HashMap;

class Bookmark implements Comparable<Bookmark> {
  static final HashMap<String, Bookmark> MODEL=new HashMap<>();
  final String url;
  final String title;
  final String id;

  static {
```

```
    add(new Bookmark("Android Developer Home",
        "https://developer.android.com",
        "687a9ea6-f0c0-448c-9cc9-a4aa6e10a1af"));
    add(new Bookmark("Android Open Source Project",
      "https://source.android.com",
      "0ee37e25-2dac-4602-8aa2-3709ac4037c8"));
    add(new Bookmark("AOSP Source Search",
      "http://xref.opersys.com/",
      "405ba533-337e-40be-abe0-fb86cd04bf7d"));
    add(new Bookmark("Stack Overflow Android Questions",
      "https://stackoverflow.com/questions/tagged/android",
      "c9599794-cb9f-46a1-ad61-971ff2a8a172"));
    add(new Bookmark("The CommonsBlog",
      "https://commonsware.com/blog/",
      "948fe25a-44d4-49d0-a23f-2783f786040d"));
    add(new Bookmark("CWAC Community",
      "https://community.commonsware.com/c/cwac",
      "4c7fac0f-fc86-4c68-8ad8-99198fc3d433"));
  }

  private static void add(Bookmark b) {
    MODEL.put(b.id, b);
  }

  Bookmark(String title, String url, String id) {
    this.url=url;
    this.title=title;
    this.id=id;
  }

  @Override
  public int compareTo(Bookmark bookmark) {
    return(title.compareTo(bookmark.title));
  }
}
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/Bookmark.java)

Here, to keep the example simple, the "database" of bookmarks is merely hardcoded roster, stored in a `HashMap`, keyed by a UUID serving as a unique identifier. In addition to its `id`, each `Bookmark` has a `title` and a `url`.

When it comes time to build an `Intent` for a given `Bookmark`, we use that `url` as the "data" facet of the `Intent`, to deliver it to our `MainActivity`:

```
  private Intent buildIntent(Bookmark item) {
    return(new Intent(getActivity(), MainActivity.class)
```

```
      .setAction("i.can.haz.reason.why.this.is.REQUIRED")
      .setData(Uri.parse(item.url)))
      .putExtra(MainActivity.EXTRA_BOOKMARK_ID, item.id)
      .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_CLEAR_TASK);
  }
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/SettingsFragment.java)

We also:

- Set the component to identify our `MainActivity`
- Set the action string because, if we do not, our app shortcut will not work
- Add the same flags to the `Intent` that are used by manifest app shortcuts, to synchronize the behavior between our one manifest app shortcut and any dynamic app shortcuts that we create
- Save the bookmark's ID in an extra

Define the Shortcuts

Android 7.1's SDK offers a `ShortcutInfo.Builder`, which lets you create `ShortcutInfo` objects, each of which represents one dynamic app shortcut.

Given a `Set` of `Bookmark` IDs, we can craft the corresponding `ShortcutInfo` objects via builders:

```
private List<ShortcutInfo> buildShortcuts(Set<String> ids) {
  List<Bookmark> items=new ArrayList<>();

  for (String id: ids) {
    items.add(Bookmark.MODEL.get(id));
  }

  if (items.size()>0) Collections.sort(items);

  List<ShortcutInfo> shortcuts=new ArrayList<>();

  for (Bookmark item : items) {
    shortcuts.add(new ShortcutInfo.Builder(getActivity(), item.id)
      .setShortLabel(item.title)
      .setIcon(buildIcon(item))
      .setIntent(buildIntent(item))
      .build());
  }
```

```
    return(shortcuts);
  }
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/SettingsFragment.java)

The `ShortcutInfo.Builder` constructor takes a `Context` for resource resolution, plus a unique ID of the app shortcut. In our case, we just use the unique ID of the `Bookmark`, since a `Bookmark` corresponds 1:1 with our dynamic app shortcuts.

The builder methods that we use here mirror the XML that we used in the manifest app widget:

Manifest App Widget XML	Builder Method
`android:shortcutShortLabel`	`setShortLabel()`
`android:icon`	`setIcon()`
`<intent>`	`setIntent()`

Our `setIntent()` call uses the `buildIntent()` method shown in the preceding section. In theory, our corresponding `buildIcon()` method would craft an icon for each bookmark, perhaps using the `favicon` of the site. Here, we just use a simple resource image, the same one for each bookmark:

```
  private Icon buildIcon(Bookmark item) {
    return(Icon.createWithResource(getActivity(),
      R.drawable.ic_bookmark_border_black_24dp));
  }
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/SettingsFragment.java)

This `buildShortcuts()` method simply creates the `ShortcutInfo` objects. To apply them, we need to get our hands on a `ShortcutManager`, via `getSystemService()`:

```
    shortcuts=getActivity().getSystemService(ShortcutManager.class);
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/SettingsFragment.java)

Then, we can call `setDynamicShortcuts()` on the `ShortcutManager`, supplying our list of `ShortcutInfo` objects, to specify that this list of `ShortcutInfo` objects represents the current roster of dynamic app shortcuts to offer to the user:

```
  private void showBookmarks() {
    updateBookmarks(bookmarks.getValues());
  }
```

```
private void updateBookmarks(Set<String> ids) {
  shortcuts.setDynamicShortcuts(buildShortcuts(ids));
}
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/SettingsFragment.java)

All of this code is appearing in a `SettingsFragment` that shows the `SwitchPreference` and `MultiSelectListPreference` for manipulating the bookmarks. `showBookmarks()` is called if the user toggles on the `SwitchPreference`, and `updateBookmarks()` is called when the user changes which items are checked in the `MultiSelectListPreference` (held in the `bookmarks` field).

Remove the Shortcuts

It is possible that you will want to remove some existing dynamic app shortcuts. In the case of the sample app, there are two possibilities:

1. The user changes the mix of bookmarks to be something other than it was before, including perhaps unchecking some previously-checked bookmarks
2. The user turns off the `SwitchPreference`, meaning that no dynamic app shortcuts should be offered

Scenario #1 is handled just by calling `setDynamicShortcuts()` on the `ShortcutManager`, as this removes any existing app shortcuts.

Scenario #2 is handled by `removeAllDynamicShortcuts()` on the `ShortcutManager`, which does pretty much what the method name suggests and removes all dynamic app shortcuts:

```
private void hideBookmarks() {
  shortcuts.removeAllDynamicShortcuts();
}
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/SettingsFragment.java)

Another option is the `removeShortcuts()` method on the `ShortcutManager`. This takes a `List` of ID values and removes those app shortcuts.

However, bear in mind that:

- You cannot remove immutable app shortcuts, so you cannot use this to remove a manifest app shortcut

- This does not affect any pinned app shortcuts, which will remain even after you have "removed" the app shortcut

What these methods are doing is changing the roster of dynamic app shortcuts that are available to the user from the launcher icon. They do not affect any existing pinned app shortcuts.

Grok the Other Verbs

`ShortcutManager` has a handful of other methods that allow you to manipulate the roster of dynamic app shortcuts that your app produces.

`addDynamicShortcuts()` will update any dynamic app shortcuts with the same IDs as the ones that you supply, and any new dynamic app shortcuts will be added. Where `setDynamicShortcuts()` says "replace the existing roster with this one", `addDynamicShortcuts()` says "update or augment the existing roster with these, and leave everything else alone".

`updateShortcuts()` is like `addDynamicShortcuts()`, except that it will only update existing dynamic app shortcuts, not add new ones.

`reportShortcutUsed()` should be called, with the ID of a shortcut, whenever that app shortcut gets used by the app. In theory, this information might help Android optimize the presentation of app shortcuts to the user, though it is unclear if this is being used at the moment. This is why we put the bookmark ID in the `EXTRA_BOOKMARK_ID` extra: so `MainActivity` can report the usage of this app shortcut.

```
    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.N_MR1) {
      String id=i.getStringExtra(EXTRA_BOOKMARK_ID);

      if (id!=null) {
        getSystemService(ShortcutManager.class)
          .reportShortcutUsed(id);
      }
    }
```

(from AppShortcuts/WeakBrowser/app/src/main/java/com/commonsware/android/appshortcuts/MainActivity.java)

(here, `i` is the `Intent` used to display this activity)

`disableShortcuts()` — where you supply it with a list of dynamic app shortcut IDs — allows you to stop pinned dynamic app shortcuts from working. While

`setDynamicShortcuts()`, `removeDynamicShortcuts()`, and `removeAllDynamicShortcuts()` affect the roster of available dynamic app shortcuts, they do not affect any pinned dynamic app shortcuts. Those will still work. If the reason why you are removing some dynamic app shortcuts is that the user is no longer eligible for those things (e.g., the user failed to renew a subscription), `disableShortcuts()` allows you to block those dynamic app shortcuts from working. The user will be shown a message instead of having the pinned dynamic app shortcut launch an activity, and you can tailor that message if desired.

Contemplate Update vs. Replace

`setDynamicShortcuts()`, `updateShortcuts()`, and `addDynamicShortcuts()` all do the same thing if there is an existing dynamic app shortcut with the same ID: update its contents to reflect whatever you passed in to those methods. This includes "updating" it to have the same information as it already has, if you have not changed anything. These not only update the roster that will be shown to the user, but they also update any pinned editions of those dynamic app shortcuts.

This introduces a potential area of confusion for the user.

For example, a Web browser that is more sophisticated than is `WeakBrowser` could keep track of which sites the user visits most often. Then, the browser could offer a dynamic app shortcut to visit that specific site. Let's pretend for a moment that, at some point in time, the user's most-visited site is the CommonsWare site. The browser would have a dynamic app shortcut for "CommonsWare", which the user could pin.

Some time later, as the user continues using the browser, the browser realizes that some new site has supplanted the CommonsWare site as the one that the user has visited the most. So, the browser will want its dynamic app shortcut roster to reflect this change.

There are two ways of going about this:

1. The browser could reuse the existing app shortcut ID (e.g., `mostPopular`) and change its label and `Intent` to reflect the new most-popular site. However, this will not only change what the user sees when looking at the available app shortcuts, but it also *changes the pinned app shortcut*. Now that home screen shortcut would bring up some other site, while the user had pinned the CommonsWare site.

2. The browser could use app shortcut IDs that are unique not for the role, but for the site (e.g., use the actual URL). If the browser uses `setDynamicShortcuts()` now, the list of shortcuts would contain one with the new app shortcut ID and would not contain one with the old app shortcut ID. The list of dynamic app shortcuts that the user sees will reflect this change. But, since the browser did not change any data with the old app shortcut ID, *the pinned one remains as it was*, still pointing to the CommonsWare site.

If this sort of thing sounds like it might be plausible for your planned use of dynamic app shortcuts, you probably want to consider the app shortcut ID to be tied to the content (e.g., the site URL), not the role the content is being applied to (e.g., the most-popular site URL).

Get the Existing Shortcuts

There are three getter methods that allow you to find out what app shortcuts are outstanding and are related to your app:

- `getDynamicShortcuts()`
- `getManifestShortcuts()`
- `getPinnedShortcuts()`

The latter, as the name suggests, lets you know which app shortcuts (manifest or dynamic) have been pinned by the user.

Note that the `ShortcutInfo` objects that you get back from these methods may not have all of their details filled in (e.g., may be missing the icon). Mostly, you will be looking for the shortcut IDs, so you can make determinations of how to manipulate the app shortcut roster (e.g., do you need to disable anything?).

Deal with Reality

Unfortunately, there are some aspects of dynamic app shortcuts which will require additional work.

If your labels for the dynamic app shortcuts have translations for other languages, you need to replace the dynamic app shortcuts when the user switches the device to a different locale. For example, you could have a manifest-registered `BroadcastReceiver`, listening for the `ACTION_LOCALE_CHANGED` system broadcast.

There, you could call `setDynamicShortcuts()` or `updateShortcuts()` to reflect the new labels.

However, outside of locale changes, you need to be careful about updating dynamic app shortcut information when your app is in the background. There is a "rate limit" established that will prevent you from changing those shortcuts too frequently. `addDynamicShortcuts()`, `setDynamicShortcuts()`, and `updateShortcuts()` each return a `boolean`; `false` indicates that your request failed due to rate-limiting. You can also call `isRateLimitingActive()` on the `ShortcutManager` to find out in advance whether your app is being rate-limited and would not be able to affect dynamic app shortcut changes.

Also, there is a limit of how many app shortcuts you can have at any one time. `geMaxShortcutCountPerActivity()` on `ShortcutManager` reports this limit. Attempting to go past that will result in an exception. Android 7.1 appears to have a limit of five; if you try enabling all six bookmarks in the sample app, you will crash.

Privacy, Security, and App Shortcuts

Bear in mind that information contained in app shortcuts will be visible to home screens and anything else using `LauncherApps` to get at the possible app shortcuts. As such, please be careful to avoid putting sensitive information in app shortcuts (e.g., labels).

PackageManager Tricks

`PackageManager` is your primary means of introspection at the component level, to determine what else is installed on the device and what components they export (activities, etc.). As such, there are many ways you can use `PackageManager` to determine if something you want is possible or not, so you can modify your behavior accordingly (e.g., disable action bar items that are not possible).

This chapter will outline some ways you can use `PackageManager` to find out what components are available to you on a device.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Asking Around

The ways to find out whether there is an activity that will respond to a given `Intent` are by means of `queryIntentActivityOptions()` and the somewhat simpler `queryIntentActivities()`.

The `queryIntentActivityOptions()` method takes the caller `ComponentName`, the "specifics" array of `Intent` instances, the overall `Intent` representing the actions you are seeking, and the set of flags. It returns a `List` of `Intent` instances matching the stated criteria, with the "specifics" ones first.

If you would like to offer alternative actions to users, but by means other than `addIntentOptions()`, you could call `queryIntentActivityOptions()`, get the `Intent` instances, then use them to populate some other user interface (e.g., a toolbar).

A simpler version of this method, `queryIntentActivities()`, is used by the `Introspection/Launchalot` sample application. This presents a "launcher" — an activity that starts other activities — but uses a `ListView` rather than a grid like the Android default home screen uses.

Here is the Java code for `Launchalot` itself:

```
package com.commonsware.android.launchalot;

import android.app.ListActivity;
import android.content.ComponentName;
import android.content.Intent;
import android.content.pm.ActivityInfo;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import java.util.Collections;
import java.util.List;

public class Launchalot extends ListActivity {
  AppAdapter adapter=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    PackageManager pm=getPackageManager();
    Intent main=new Intent(Intent.ACTION_MAIN, null);

    main.addCategory(Intent.CATEGORY_LAUNCHER);

    List<ResolveInfo> launchables=pm.queryIntentActivities(main, 0);

    Collections.sort(launchables,
                     new ResolveInfo.DisplayNameComparator(pm));
```

```
  adapter=new AppAdapter(pm, launchables);
  setListAdapter(adapter);
}

@Override
protected void onListItemClick(ListView l, View v,
                               int position, long id) {
  ResolveInfo launchable=adapter.getItem(position);
  ActivityInfo activity=launchable.activityInfo;
  ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                       activity.name);
  Intent i=new Intent(Intent.ACTION_MAIN);

  i.addCategory(Intent.CATEGORY_LAUNCHER);
  i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
             Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);
  i.setComponent(name);

  startActivity(i);
}

class AppAdapter extends ArrayAdapter<ResolveInfo> {
  private PackageManager pm=null;

  AppAdapter(PackageManager pm, List<ResolveInfo> apps) {
    super(Launchalot.this, R.layout.row, apps);
    this.pm=pm;
  }

  @Override
  public View getView(int position, View convertView,
                      ViewGroup parent) {
    if (convertView==null) {
      convertView=newView(parent);
    }

    bindView(position, convertView);

    return(convertView);
  }

  private View newView(ViewGroup parent) {
    return(getLayoutInflater().inflate(R.layout.row, parent, false));
  }

  private void bindView(int position, View row) {
    TextView label=(TextView)row.findViewById(R.id.label);
```

```
      label.setText(getItem(position).loadLabel(pm));

      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      icon.setImageDrawable(getItem(position).loadIcon(pm));
    }
  }
}
```

(from Introspection/Launchalot/app/src/main/java/com/commonsware/android/launchalot/Launchalot.java)

In `onCreate()`, we:

1. Get a `PackageManager` object via `getPackageManager()`
2. Create an `Intent` for `ACTION_MAIN` in `CATEGORY_LAUNCHER`, which identifies activities that wish to be considered "launchable"
3. Call `queryIntentActivities()` to get a `List` of `ResolveInfo` objects, each one representing one launchable activity
4. Sort those `ResolveInfo` objects via a `ResolveInfo.DisplayNameComparator` instance
5. Pour them into a custom `AppAdapter` and set that to be the contents of our `ListView`

`AppAdapter` is an `ArrayAdapter` subclass that maps the icon and name of the launchable `Activity` to a row in the `ListView`, using a custom row layout.

Finally, in `onListItemClick()`, we construct an `Intent` that will launch the clicked-upon `Activity`, given the information from the corresponding `ResolveInfo` object. Not only do we need to populate the `Intent` with `ACTION_MAIN` and `CATEGORY_LAUNCHER`, but we also need to set the component to be the desired `Activity`. We also set `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` flags, following Android's own launcher implementation from the Home sample project. Finally, we call `startActivity()` with that `Intent`, which opens up the activity selected by the user.

The result is a simple list of launchable activities:

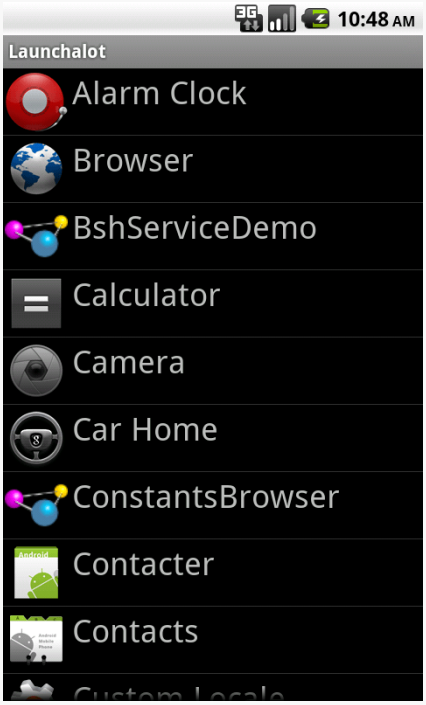


Figure 898: The Launchalot sample application

There is also a `resolveActivity()` method that takes a template Intent, as do `queryIntentActivities()` and `queryIntentActivityOptions()`. However, `resolveActivity()` returns the single best match, rather than a list.

NOTE: On modern versions of Android, there is a `LauncherApps` class that simplifies a lot of this and takes things like Android Work profiles into account. For really implementing a home screen-style launcher, you will probably want to use `LauncherApps`. However, using `PackageManager` to find what can handle certain `Intent` structures is used for other purposes beyond home screen launchers.

Preferred Activities

Users, when presented with a default activity chooser, usually have the option to make their next choice be the default for this action for now on. The next time they do whatever they did to bring up the chooser, it should go straight to this default. This is known in the system as the "preferred activity" for an `Intent` structure, and is stored in the system as a set of pairs of `IntentFilter` objects and the corresponding `ComponentName` of the preferred activity.

To find out what the preferred activities are on a given device, you can ask `PackageManager` to `getPreferredActivities()`. You pass in a `List<IntentFilter>` and a `List<ComponentName>`, and Android fills in those lists with the preferred activity information.

To see this in action, take a look at the `Introspection/PrefActivities` sample application. This simply loads all of the information into a `ListView`, using `android.R.layout.simple_list_item_2` as a row layout for a title-and-description pattern.

The `PackageManager` logic is confined to `onCreate()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  PackageManager mgr=getPackageManager();

  mgr.getPreferredActivities(filters, names, null);
  setListAdapter(new IntentFilterAdapter());
}
```

(from Introspection/PrefActivities/app/src/main/java/com/commonsware/android/prefact/PreferredActivitiesDemoActivity.java)

In this case, the two lists are data members of the activity:

```
ArrayList<IntentFilter> filters=new ArrayList<IntentFilter>();
ArrayList<ComponentName> names=new ArrayList<ComponentName>();
```

(from Introspection/PrefActivities/app/src/main/java/com/commonsware/android/prefact/PreferredActivitiesDemoActivity.java)

Most of the logic is in formatting the `ListView` contents. `IntentFilter`, unfortunately, does not come with a method that gives us a human-readable dump of its definition. As a result, we need to roll that ourselves. Compounding the problem is that `IntentFilter` tends to return `Iterator` objects for its collections (e.g., roster of actions), rather than something `Iterable`. The activity leverages an `Iterator`-to-`Iterable` wrapper culled from a Stack Overflow answer to help with this. The `IntentFilterAdapter` and helper code looks like this:

```
// from http://stackoverflow.com/a/8555153/115145

public static <T> Iterable<T> in(final Iterator<T> iterator) {
  class SingleUseIterable implements Iterable<T> {
    private boolean used=false;
```

```
    @Override
    public Iterator<T> iterator() {
      if (used) {
        throw new IllegalStateException("Already invoked");
      }
      used=true;
      return iterator;
    }
  }
  return new SingleUseIterable();
}

class IntentFilterAdapter extends ArrayAdapter<IntentFilter> {
  IntentFilterAdapter() {
    super(PreferredActivitiesDemoActivity.this,
          android.R.layout.simple_list_item_2, android.R.id.text1,
          filters);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    TextView filter=(TextView)row.findViewById(android.R.id.text1);
    TextView name=(TextView)row.findViewById(android.R.id.text2);

    filter.setText(buildTitle(getItem(position)));
    name.setText(names.get(position).getClassName());

    return(row);
  }

  String buildTitle(IntentFilter filter) {
    StringBuilder buf=new StringBuilder();
    boolean first=true;

    if (filter.countActions() > 0) {
      for (String action : in(filter.actionsIterator())) {
        if (first) {
          first=false;
        }
        else {
          buf.append('/');
        }

        buf.append(action.replaceAll("android.intent.action.", ""));
      }
    }
```

```
    if (filter.countDataTypes() > 0) {
      first=true;

      for (String type : in(filter.typesIterator())) {
        if (first) {
          buf.append(" : ");
          first=false;
        }
        else {
          buf.append('|');
        }

        buf.append(type);
      }
    }

    if (filter.countDataSchemes() > 0) {
      buf.append(" : ");
      buf.append(filter.getDataScheme(0));

      if (filter.countDataSchemes() > 1) {
        buf.append(" (other schemes)");
      }
    }

    if (filter.countDataPaths() > 0) {
      buf.append(" : ");
      buf.append(filter.getDataPath(0));

      if (filter.countDataPaths() > 1) {
        buf.append(" (other paths)");
      }
    }

    return(buf.toString());
  }
}
```

(from Introspection/PrefActivities/app/src/main/java/com/commonsware/android/prefact/PreferredActivitiesDemoActivity.java)

The resulting activity shows a simple description of the `IntentFilter` along with the class name of the corresponding activity in each row:

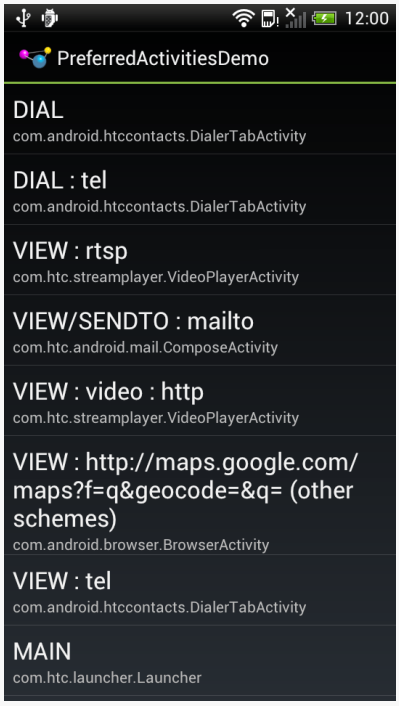


Figure 899: Preferred Activities on a Stock HTC One S

Another way to think about preferred activities is to determine what specific activity will handle a `startActivity()` call on some `Intent`. If there is only one alternative, or the user chose a preferred activity, that activity should handle the `Intent`. Otherwise, the activity handling the `Intent` *should* be one implementing a chooser. The `resolveActivity()` method on `PackageManager` can let us know what will handle the `Intent`.

To examine what `resolveActivity()` returns, take a look at the `Introspection/Resolver` sample application.

The activity — which uses `Theme.Translucent.NoTitleBar` and so has no UI of its own — is fairly short:

```
package com.commonsware.android.resolver;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.net.Uri;
import android.os.Bundle;
```

```
import android.widget.Toast;

public class ResolveActivityDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PackageManager mgr=getPackageManager();
    Intent i=
        new Intent(Intent.ACTION_VIEW,
                   Uri.parse("https://commonsware.com"));
    ResolveInfo ri=
        mgr.resolveActivity(i, PackageManager.MATCH_DEFAULT_ONLY);

    Toast.makeText(this, ri.loadLabel(mgr), Toast.LENGTH_LONG).show();

    startActivity(i);
    finish();
  }
}
```

(from Introspection/Resolver/app/src/main/java/com/commonsware/android/resolver/ResolveActivityDemoActivity.java)

We get a `PackageManager`, create an `Intent` to test, and pass the `Intent` to `resolveActivity()`. We include `MATCH_DEFAULT_ONLY` so we only get activities that have `CATEGORY_DEFAULT` in their `<intent-filter>` elements. We then use `loadLabel()` on the resulting `ResolveInfo` object to get the display name of the activity, toss that in a `Toast`, and invoke `startActivity()` on the `Intent` to confirm the results.

On a device with only one option, or with a default chosen, the `Toast` will show the name of the preferred activity (e.g., Browser). On most devices with more than one option, the `startActivity()` call will display a chooser, and the `Toast` will show the display name of the chooser (e.g., "Android System").

However, on some devices — notably newer models from HTC distributed in the US — `resolveActivity()` indicates that `HTCLinkifyDispatcher` is the one that will handle `ACTION_VIEW` on a URL... even if there is more than one browser installed and no default has been specified. This is part of a workaround that HTC added in 2012 to help deal with a patent dispute with Apple.

Middle Management

The `PackageManager` class offers much more than merely `queryIntentActivities()` and `queryIntentActivityOptions()`. It is your gateway to all sorts of analysis of what is installed and available on the device where your application is installed and available. If you want to be able to intelligently connect to third-party applications based on whether or not they are around, `PackageManager` is what you will want.

Finding Applications and Packages

Packages are what get installed on the device — a package is the in-device representation of an APK. An application is defined within a package's manifest. Between the two, you can find out all sorts of things about existing software installed on the device.

Specifically, `getInstalledPackages()` returns a `List` of `PackageInfo` objects, each of which describes a single package. Here, you can find out:

1. The version of the package, in terms of a monotonically increasing number (`versionCode`) and the display name (`versionName`)
2. Details about all of the components — activities, services, etc. — offered by this package
3. Details about the permissions the package requires

Similarly, `getInstalledApplications()` returns a `List` of `ApplicationInfo` objects, each providing data like:

1. The user ID that the application will run as
2. The path to the application's private data directory
3. Whether or not the application is enabled

In addition to those methods, you can call:

1. `getApplicationIcon()` and `getApplicationLabel()` to get the icon and display name for an application
2. `getLaunchIntentForPackage()` to get an `Intent` for something launchable within a named package
3. `setApplicationEnabledSetting()` to enable or disable an application

Finding Resources

You can access resources from another application, apparently without any security restrictions. This may be useful if you have multiple applications and wish to share resources for one reason or another.

The `getResourcesForActivity()` and `getResourcesForApplication()` methods on `PackageManager` return a `Resources` object. This is just like the one you get for your own application via `getResources()` on any `Context` (e.g., `Activity`). However, in this case, you identify what activity or application you wish to get the `Resources` from (e.g., supply the application's package name as a `String`).

There are also `getText()` and `getXml()` methods that dive into the `Resources` object for an application and pull out specific `String` or `XmlPullParser` objects. However, these require you to know the resource ID of the resource to be retrieved, and that may be difficult to manage between disparate applications.

Finding Components

Not only does Android offer "query" and "resolve" methods to find activities, but it offers similar methods to find other sorts of Android components:

1. `queryBroadcastReceivers()`
2. `queryContentProviders()`
3. `queryIntentServices()`
4. `resolveContentProvider()`
5. `resolveService()`

For example, you could use `resolveService()` to determine if a certain remote service is available, so you can disable certain UI elements if the service is not on the device. You could achieve the same end by calling `bindService()` and watching for a failure, but that may be later in the application flow than you would like.

There is also a `setComponentEnabledSetting()` to toggle a component (activity, service, etc.) on and off. While this may seem esoteric, there are a number of possible uses for this method, such as:

1. Flagging a launchable activity as disabled in your manifest, then enabling it programmatically after the user has entered a license key, achieved some level or standing in a game, or any other criteria

2. Controlling whether a `BroadcastReceiver` registered in the manifest is hooked into the system or not, replicating the level of control you have with `registerReceiver()` while still taking advantage of the fact that a manifest-registered `BroadcastReceiver` can be started even if no other component of your application is running

Remote Services and the Binding Pattern

Earlier in this book, we covered using services by sending commands to them to be processed. That "command pattern" is one of two primary means of interacting with a service — the binding pattern is the other. With the binding pattern, your service exposes a more traditional API, in the form of a "binder" object with methods of your choosing. On the plus side, you get a richer interface. However, it more tightly ties your activity to your service, which may cause you problems with configuration changes.

Either the command pattern or the binding pattern can be used, if desired, across process boundaries, with the client being some third-party application. In either case, you will need to export your service via an `<intent-filter>`. And, in the case of the binding pattern, your "binder" implementation will have some restrictions.

This chapter covers the binding pattern for local services, plus inter-process commands and binding (a.k.a., remote services).

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- broadcast Intents
- service theory

The Binding Pattern

Implementing the binding pattern requires work on both the service side and the client side. The service will need to have a full implementation of the `onBind()` method, which typically just returns `null` or throws some sort of runtime exception for a service solely implementing the command pattern. And, the client (e.g., an activity) will need to ask to bind to the service, instead of (or perhaps in addition to) starting the service.

What the Service Does

The service implements a subclass of `Binder` that represents the service's exposed API. For a local service, your `Binder` can have pretty much whatever methods you want: method names, parameters, return types, and exceptions thrown are up to you. When you get into remote services, your `Binder` implementation will be substantially more constrained, to support inter-process communication.

Then, your `onBind()` method returns an instance of the `Binder`.

What the Client Does

Clients call `bindService()`, supplying the `Intent` that identifies the service, a `ServiceConnection` object representing the client side of the binding, and an optional `BIND_AUTO_CREATE` flag. As with `startService()`, `bindService()` is asynchronous. The client will not know anything about the status of the binding until the `ServiceConnection` object is called with `onServiceConnected()`. This not only indicates the binding has been established, but for local services it provides the Binder object that the service returned via `onBind()`. At this point, the client can use the `Binder` to ask the service to do work on its behalf.

Note that if the service is not already running, and if you provide `BIND_AUTO_CREATE`, then the service will be created first before being bound to the client. If you skip `BIND_AUTO_CREATE`, and the service is not already running, `bindService()` is supposed to return `false`, indicating there was no existing service to bind to. However, in actuality, Android returns `true`, due to an apparent bug.

Eventually, the client will need to call `unbindService()`, to indicate it no longer needs to communicate with the service. For example, an activity might call `bindService()` in its `onCreate()` method, then call `unbindService()` in its `onDestroy()` method. Once you call `unbindService()`, your `Binder` object is no

longer safe to be used by the client. If there are no other bound clients to the service, Android will shut down the service as well, releasing its memory. Hence, we do not need to call `stopService()` ourselves — Android handles that, if needed, as a side effect of unbinding.

Your `ServiceConnection` object will also need an `onServiceDisconnected()` method. This will be called only if there is an unexpected disconnection, such as the service crashing with an unhandled exception.

A Binding Sample

In the chapter introducing services, we saw a sample app that would download a file off of a Web server. That sample used the command pattern, telling the service what to download via an `Intent` extra. In this chapter, we will review a few variations of that sample, all of which use the binding pattern instead of the command pattern.

Right now, we are focused on local services, and so the `Binding/Local` sample project does the download via a local bound service.

We start by defining an interface that will serve as the "contract" between the client (fragment) and service. This interface, `IDownload`, contains a single `download()` method:

```
package com.commonsware.android.advservice.binding;

// Declare the interface.
interface IDownload {
  void download(String url);
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/IDownload.java)

Our service, `DownloadService`, implements just one method, `onBind()`, which returns an instance of a `DownloadBinder`:

```
package com.commonsware.android.advservice.binding;

import android.app.Service;
import android.content.Intent;
import android.net.Uri;
import android.os.Binder;
import android.os.Environment;
import android.os.IBinder;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class DownloadService extends Service {
  @Override
  public IBinder onBind(Intent intent) {
    return(new DownloadBinder());
  }

  private static class DownloadBinder extends Binder implements IDownload {
    @Override
    public void download(String url) {
      new DownloadThread(url).start();
    }
  }

  private static class DownloadThread extends Thread {
    String url=null;

    DownloadThread(String url) {
      this.url=url;
    }

    @Override
    public void run() {
      try {
        File root=
            Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

        root.mkdirs();

        File output=new File(root, Uri.parse(url).getLastPathSegment());

        if (output.exists()) {
          output.delete();
        }

        HttpURLConnection c=(HttpURLConnection)new URL(url).openConnection();

        FileOutputStream fos=new FileOutputStream(output.getPath());
        BufferedOutputStream out=new BufferedOutputStream(fos);

        try {
          InputStream in=c.getInputStream();
          byte[] buffer=new byte[8192];
          int len=0;

          while ((len=in.read(buffer)) >= 0) {
            out.write(buffer, 0, len);
          }

          out.flush();
        }
        finally {
          fos.getFD().sync();
          out.close();
          c.disconnect();
```

```
      }
    }
    catch (IOException e2) {
      Log.e("DownloadJob", "Exception in download", e2);
    }
  }
 }
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadService.java)

`DownloadBinder` implements the `IDownload` interface. Its `download()` method, in turn, forks a `DownloadThread` to perform the download in the background — remember, for local services, the methods you invoke on the `Binder` are executed on whatever thread you call them on.

Our fragment, `DownloadFragment`, loads our layout, `res/layout/main.xml`, containing a `Button` to trigger the download:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/go"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
    android:text="@string/go"/>
```

(from Binding/Local/app/src/main/res/layout/main.xml)

The implementation of `onCreateView()` simply loads that layout, gets the `Button`, sets up the fragment as being the click listener for the `Button`, and disables the `Button`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, container, false);

    btn=(Button)result.findViewById(R.id.go);
    btn.setOnClickListener(this);
    btn.setEnabled(binding!=null);

    return(result);
  }
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadFragment.java)

The reason why we disable the `Button` is because we are not connected to our service at this point, and until we are, we cannot allow the user to try to download a file.

In `onCreate()` of our fragment, we mark the fragment as retained and bind to the service:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  appContext=(Application)getActivity().getApplicationContext();
  appContext.bindService(new Intent(getActivity(),
      DownloadService.class),
    this, Context.BIND_AUTO_CREATE);
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadFragment.java)

You will notice something curious here: `getApplicationContext()`. Technically, we could bind to the service directly from the `Activity`, by calling `bindService()` on it, as `bindService()` is a method on `Context`. However, our service binding represents some state, and it is possible that this state will hold a reference to the `Context` that created the binding. In that case, we run the risk of leaking our original activity during a configuration change. The `getApplicationContext()` method returns the global `Application` singleton, which is a `Context` suitable for binding, but one that cannot be leaked, since it is already in a global scope. In effect, it is "pre-leaked".

The call to `setRetainInstance()` allows the fragment – serving as our `ServiceConnection` — to survive a configuration change, so we can cleanly unbind from the service later on, when `onDestroy()` is called.

Some time after `onCreate()` is called and we call `bindService()`, our `onServiceConnected()` method will be called, as we designated our fragment to be the `ServiceConnection`. Here, we can cast the `IBinder` object we receive to be our `IDownload` interface to the service, and we can enable the `Button`:

```
@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
  binding=(IDownload)binder;
  btn.setEnabled(true);
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadFragment.java)

Since we are implementing the `ServiceConnection` interface, our fragment also needs to implement the `onServiceDisconnected()` method, invoked if our service crashes. Here, we delegate responsibility to a `disconnect()` private method, which removes our link to the `IDownload` object and disables our `Button`:

```
@Override
public void onServiceDisconnected(ComponentName className) {
  disconnect();
}

private void disconnect() {
  binding=null;
  btn.setEnabled(false);
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadFragment.java)

And, when our fragment is destroyed, we unbind from the service (using the same `Context` as before, from `getApplicationContext()`) and `disconnect()`:

```
@Override
public void onDestroy() {
  appContext.unbindService(this);
  disconnect();

  super.onDestroy();
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadFragment.java)

However, in between `onServiceConnected()` and either `onServiceDisconnected()` or `onDestroy()`, the user can click the `Button`, which will trigger the download via a call to `download()` on our `IDownload` instance:

```
@Override
public void onClick(View view) {
  binding.download(TO_DOWNLOAD);
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadFragment.java)

The `DownloadBindingDemo` activity adds our `DownloadFragment` via a `FragmentTransaction`:

```
package com.commonsware.android.advservice.binding;

import android.app.Activity;
import android.os.Bundle;

public class DownloadBindingDemo extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                                  .add(android.R.id.content,
                                       new DownloadFragment()).commit();
    }
  }
}
```

(from Binding/Local/app/src/main/java/com/commonsware/android/advservice/binding/DownloadBindingDemo.java)

Starting and Binding

Some developers will use *both* `startService()` and `bindService()` at the same time. The typical argument is that they need frequent updates from the service (e.g., percentage of progress, for updating a `ProgressBar`) in the client and are concerned about the overhead of sending broadcasts.

With the advent of `LocalBroadcastManager` and other event bus implementations, binding to a service you are using with `startService()` should no longer be necessary.

When IPC Attacks!

If you wish to extend the binding pattern to serve in the role of IPC, whereby other processes can get at your `Binder` and call its methods, you will need to use AIDL: the Android Interface Description Language. If you have used IPC mechanisms like SOAP, XML-RPC, DCOM, CORBA, or the like, you will recognize the notion of IDL. AIDL describes the public IPC interface, and Android supplies tools to build the client and server side of that interface.

With that in mind, let's take a look at AIDL and IPC.

Write the AIDL

IDLs are frequently written in a "language-neutral" syntax. AIDL, on the other hand, looks a lot like a Java interface file. For example, here is some AIDL:

```
package com.commonsware.android.advservice.remotebinding;

// Declare the interface.
interface IDownload {
  void download(String url);
}
```

(from Binding/Remote/Service/app/src/main/aidl/com/commonsware/android/advservice/remotebinding/IDownload.aidl)

As you will notice, this looks suspiciously like the regular Java interface we used in the simple binding example earlier in this chapter.

As with a Java interface, you declare a package at the top. As with a Java interface, the methods are wrapped in an interface declaration (`interface IDownload { ... }`). And, as with a Java interface, you list the methods you are making available.

The differences, though, are critical.

First, not every Java type can be used as a parameter. Your choices are:

1. Primitive values (`int`, `float`, `double`, `boolean`, etc.)
2. `String` and `CharSequence`
3. `List` and `Map` (from `java.util`)
4. Any other AIDL-defined interfaces
5. Any Java classes that implement the `Parcelable` or `Serializable` interface

In the case of the latter two categories, you need to include `import` statements referencing the names of the classes or interfaces that you are using (e.g., `import com.commonsware.android.ISomething`). This is true even if these classes are in your own package — you have to import them anyway.

Next, parameters can be classified as `in`, `out`, or `inout`. Values that are `out` or `inout` can be changed by the service and those changes will be propagated back to the client. Primitives (e.g., `int`) can only be `in`.

Also, you cannot throw any exceptions. You will need to catch all exceptions in your code, deal with them, and return failure indications some other way (e.g., error code return values).

Name your AIDL files with the `.aidl` extension and place them in the proper directory based on the package name:

- For native Android Studio projects, this will be an `aidl/` directory in your `src/` source set, as a peer of your `java/` directory, with the same sort of subdirectories-based-on-the-Java-package approach as you use for regular Java source code
- For Eclipse-compatible projects, the `.aidl` files will go alongside your `.java` files in the `src/` directory tree

When you build your project, either via an IDE or via command-line build tools, the `aidl` utility from the Android SDK will translate your AIDL into a server stub and a client proxy.

Implement the Interface

Given the AIDL-created server stub, now you need to implement the service, either directly in the stub, or by routing the stub implementation to other methods you have already written.

The mechanics of this are fairly straightforward:

1. Create a subclass of the AIDL-generated `.Stub` class (e.g., `IDownload.Stub`)
2. Implement methods matching up with each of the methods you placed in the AIDL
3. Return an instance of this subclass from your `onBind()` method in the `Service` subclass

Note that AIDL IPC calls are synchronous, and so the caller is blocked until the IPC method returns. Hence, your services need to be quick about their work.

We will see examples of service stubs later in this chapter.

Service From Afar

So, given our AIDL description, let us examine a sample implementation, using AIDL for a remote service.

Our sample applications — shown in the `Binding/Remote/Service` and `Binding/Remote/Client` sample projects — simply move the service logic into a separate project from the client logic.

Service Names

To bind to a service's AIDL-defined API, you need to craft an `Intent` that can identify the service in question. In the case of a local service, that Intent can use the local approach of directly referencing the service class.

Obviously, that is not possible in a remote service case, where the service class is not in the same process, and may not even be known by name to the client.

When you define a service to be used by remote, you need to add an `<intent-filter>` element to your service declaration in the manifest, indicating how you want that service to be referred to by clients. The manifest for `RemoteService` is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.advservice.remotebinding.svc"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14" />

  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <service android:name=".DownloadService">
      <intent-filter>
        <action android:name="com.commonsware.android.advservice.remotebinding.IDownload" />
      </intent-filter>
    </service>
  </application>

</manifest>
```

(from Binding/Remote/Service/app/src/main/AndroidManifest.xml)

Here, we say that the service can be identified by the name `com.commonsware.android.advservice.IDownload`. So long as the client uses this name to identify the service, it can bind to that service's API.

In this case, the name is not an implementation, but the AIDL API, as you will see below. In effect, this means that so long as some service exists on the device that implements this API, the client will be able to bind to something.

Remote Services and Implicit Intents

We are used to a device having multiple activities that can respond to the same `<intent-filter>`. In that case, by default, the user will see a chooser if we try to start one of those activities.

We are used to a device having multiple `BroadcastReceiver` components that can respond to the same `<intent-filter>` (or `IntentFilter`). In that case, in a regular broadcast, all eligible receivers will receive it.

We are used to it being impossible to have multiple `ContentProvider` components with the same authority, as the second one fails on install with an `INSTALL_FAILED_CONFLICTING_PROVIDER` error.

What happens if there are *two* (or more) services installed on the device that claim to support the same `<intent-filter>`, but have different package names? You might think that this would fail on install, as happens with providers with duplicate authorities. Alas, it does not... prior to Android 5.0. Instead, the higher-priority `<intent-filter>` gets it (set via the `android:priority` attribute). If 2+ implementations have the same priority, the first one installed wins.

So, if we have `BadService` and `GoodService`, both responding to the same `<intent-filter>`, and a client app tries to communicate to `GoodService` via the implicit `Intent` matching that `<intent-filter>`, it might *actually* be communicating with `BadService`, simply because `BadService` was installed first. The user is oblivious to this.

Android 5.0 solves this by preventing binding using an implicit `Intent`. This, however, presents a conundrum:

- We cannot bind using an implicit `Intent`

- We do not know how to construct an explicit `Intent` identifying the desired service, as that might be from a third-party app

As you will see, when we examine the client side of this sample, we have to use `PackageManager` to convert an implicit `Intent` into a valid explicit `Intent` for our service. This not only allows us to comply with the Android 5.0 binding restriction, but it gives us an opportunity to detect and handle the cases where there is no matching service (e.g., the service app has not yet been installed) or when there is more than one matching service (e.g., `BadService` and `GoodService`). And the techniques that all of this uses works on pretty much any version of Android, so while we *need* them for Android 5.0 and higher, we can *use* them anywhere.

The Service

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, `IDownload`:

```
package com.commonsware.android.advservice.remotebinding;

// Declare the interface.
interface IDownload {
  void download(String url);
}
```

(from Binding/Remote/Service/app/src/main/aidl/com/commonsware/android/advservice/remotebinding/IDownload.aidl)

And there is the actual service class itself, `DownloadService`:

```
package com.commonsware.android.advservice.remotebinding.svc;

import android.app.Service;
import android.content.Intent;
import android.net.Uri;
import android.os.Environment;
import android.os.IBinder;
import android.util.Log;
import com.commonsware.android.advservice.remotebinding.IDownload;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class DownloadService extends Service {
  @Override
  public IBinder onBind(Intent intent) {
    return(new DownloadBinder());
```

```
  }

  private static class DownloadBinder extends IDownload.Stub {
    @Override
    public void download(String url) {
      new DownloadThread(url).start();
    }
  }

  private static class DownloadThread extends Thread {
    String url=null;

    DownloadThread(String url) {
      this.url=url;
    }

    @Override
    public void run() {
      try {
        File root=
            Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

        root.mkdirs();

        File output=new File(root, Uri.parse(url).getLastPathSegment());

        if (output.exists()) {
          output.delete();
        }

        HttpURLConnection c=(HttpURLConnection)new URL(url).openConnection();

        FileOutputStream fos=new FileOutputStream(output.getPath());
        BufferedOutputStream out=new BufferedOutputStream(fos);

        try {
          InputStream in=c.getInputStream();
          byte[] buffer=new byte[8192];
          int len=0;

          while ((len=in.read(buffer)) >= 0) {
            out.write(buffer, 0, len);
          }

          out.flush();
        }
        finally {
          fos.getFD().sync();
          out.close();
          c.disconnect();
        }
      }
      catch (IOException e2) {
        Log.e("DownloadJob", "Exception in download", e2);
      }
    }
  }
}
```

(from Binding/Remote/Service/app/src/main/java/com/commonsware/android/advservice/remotebinding/svc/DownloadService.java)

This is identical to the local binding example, with one key difference: `DownloadBinder` now extends `IDownload.Stub` rather than the generic `Binder` class.

The Client

The client — a revised version of `DownloadFragment` — connects to the remote service to ask it to download the file on the user's behalf. This has three changes of note over our original local implementation.

First, when we call `download()` on the `IDownload` object, we need to catch a `RemoteException`. This will be thrown if the service crashes during our request or otherwise is unable to return properly:

```
@Override
public void onClick(View view) {
  try {
    binding.download(TO_DOWNLOAD);
  }
  catch (RemoteException e) {
    Log.e(getClass().getSimpleName(), "Exception requesting download", e);
    Toast.makeText(getActivity(), e.getMessage(), Toast.LENGTH_LONG).show();
  }
}
```

(from Binding/Remote/Client/app/src/main/java/com/commonsware/android/advservice/remotebinding/client/DownloadFragment.java)

Second, our `onServiceConnected()` uses `IDownload.Stub.asInterface()` to convert the raw `IBinder` into an `IDownload` object for use:

```
@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
  binding=IDownload.Stub.asInterface(binder);
  btn.setEnabled(true);
}
```

(from Binding/Remote/Client/app/src/main/java/com/commonsware/android/advservice/remotebinding/client/DownloadFragment.java)

Third, our binding logic in `onCreate()` is significantly more complicated:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);
```

```
  appContext=(Application)getActivity().getApplicationContext();

  Intent implicit=new Intent(IDownload.class.getName());
  List<ResolveInfo> matches=getActivity().getPackageManager()
    .queryIntentServices(implicit, 0);

  if (matches.size() == 0) {
    Toast.makeText(getActivity(), "Cannot find a matching service!",
      Toast.LENGTH_LONG).show();
  }
  else if (matches.size() > 1) {
    Toast.makeText(getActivity(), "Found multiple matching services!",
      Toast.LENGTH_LONG).show();
  }
  else {
    Intent explicit=new Intent(implicit);
    ServiceInfo svcInfo=matches.get(0).serviceInfo;
    ComponentName cn=new ComponentName(svcInfo.applicationInfo.packageName,
      svcInfo.name);

    explicit.setComponent(cn);
    appContext.bindService(explicit, this, Context.BIND_AUTO_CREATE);
  }
}
```

(from Binding/Remote/Client/app/src/main/java/com/commonsware/android/advservice/remotebinding/client/DownloadFragment.java)

Here, we:

- Get the `Application` singleton `Context` as before
- Craft an implicit `Intent` for the service, using the appropriate action string (which, in this case, happens to be the fully-qualified name of the `IDownload` interface)
- Use `PackageManager` and `queryIntentServices()` to find out all services that implement a matching `<intent-filter>` for that implicit `Intent`
- Fail with a `Toast` if there is not exactly one such service
- Use the `ServiceInfo` object from our `queryIntentServices()` call to craft an explicit `Intent`, with the same structure as the implicit `Intent` had, but also with the actual matched component (via `setComponent()`)
- Use the explicit `Intent` to bind to the service

Note that the client needs its own copy of `IDownload.aidl`. After all, it is a totally separate application, and therefore does not share source code with the service.

If you compile both applications and upload them to the device, then start up the client, you can have the service download the file.

Tightening Up the Security

The previous sample confirms that there is exactly one service that matches the desired `Intent`. This catches the zero-service scenario (requiring the user to install the other app) and catches the multiple-service scenario (where one service is an attacker, presumably).

However, what happens if there is only one service installed, and it is not the desired service, but rather is an attacker? The preceding binding code will still go ahead and bind with that service.

You might consider just examining the package name/application ID of the other service, to see if it matches an expected value. However, that will not help you if the attacker is a modified version of the real service, one that kept its original package name but changed the service to do evil things.

Checking the digital signature of the other service is a more robust check, as that cannot readily be forged. Even if somebody modifies and repackages the app with the service, that app would wind up being signed by a different signing key, which you can detect.

Moreover, this approach can be used in both directions: the client can validate the service, and the service can validate the client. For example, perhaps as part of a licensing scheme, your service can only be used by apps developed by certain firms, based upon their signing keys.

The `Binding/SigCheck/Client` sample project illustrates a client that will perform this signature check on the client side. The corresponding service project – `Binding/SigCheck/Service` – will perform a signature check on the service side.

Adding the Dependency

Both projects use the CWAC-Security library, described elsewhere in this book, to do the signature checking. Hence, their Gradle build files have a dependency on that library:

```
repositories {
    maven {
```

```
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
  implementation 'com.android.support:support-v13:21.0.3'
    implementation 'com.commonsware.cwac:security:0.4.+'
}
```

(from Binding/SigCheck/Client/app/build.gradle)

Adding the Signature Check: Client

The client's `DownloadFragment` is nearly the same as before, with an adjustment to `onCreate()` to check the signature if there is exactly one service that matches the `Intent`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  appContext=(Application)getActivity().getApplicationContext();

  Intent implicit=new Intent(IDownload.class.getName());
  List<ResolveInfo> matches=getActivity().getPackageManager()
    .queryIntentServices(implicit, 0);

  if (matches.size() == 0) {
    Toast.makeText(getActivity(), "Cannot find a matching service!",
      Toast.LENGTH_LONG).show();
  }
  else if (matches.size() > 1) {
    Toast.makeText(getActivity(), "Found multiple matching services!",
      Toast.LENGTH_LONG).show();
  }
  else {
    ServiceInfo svcInfo=matches.get(0).serviceInfo;

    try {
      String otherHash=SignatureUtils.getSignatureHash(getActivity(),
        svcInfo.applicationInfo.packageName);
      String expected=getActivity().getString(R.string.expected_sig_hash);

      if (expected.equals(otherHash)) {
        Intent explicit=new Intent(implicit);
        ComponentName cn=new ComponentName(svcInfo.applicationInfo.packageName,
          svcInfo.name);

        explicit.setComponent(cn);
        appContext.bindService(explicit, this, Context.BIND_AUTO_CREATE);
      }
      else {
```

```
        Toast.makeText(getActivity(), "Unexpected signature found!",
          Toast.LENGTH_LONG).show();
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception trying to get signature hash", e);
    }
  }
}
```

(from Binding/SigCheck/Client/app/src/main/java/com/commonsware/android/advservice/remotebinding/sigcheck/DownloadFragment.java)

In the one-match scenario, we get the signature of the other app, by using `getSignatureHash()` on `SignatureUtils`, passing in the package name of the other app. We then compare that with a hard-coded expected hash, pulled from a string resource, one that is unfortunately too long to represent in this book.

Only if those two match do we go ahead with the binding.

Adding the Signature Check: Service

This gets a bit more complicated, as we first need to figure out who the client *is*, before we can validate the signature. In the case of the client connecting to the service, we know the application ID of the service courtesy of the `queryIntentServices()` call. On the service side, we need to use a different approach to identify who the client is.

To do this work, `DownloadBinder` now needs a `Context` with which to work, so `onBind()` passes one to a revised `DownloadBinder` constructor:

```
@Override
public IBinder onBind(Intent intent) {
  return(new DownloadBinder(this));
}
```

(from Binding/SigCheck/Service/app/src/main/java/com/commonsware/android/advservice/remotebinding/sig/DownloadService.java)

The constructor holds on to three things:

- a `Context`, in this case the `Application` obtained from the `Service`
- a `PackageManager`, as we will need this for the signature lookup
- the expected hash of the client's signing key, pulled once again from a string resource

```
private static class DownloadBinder extends IDownload.Stub {
  private final PackageManager pm;
  private final String expectedHash;
  private final Context ctxt;

  public DownloadBinder(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
    this.pm=this.ctxt.getPackageManager();
    this.expectedHash=this.ctxt.getString(
      R.string.expected_sig_hash);
  }
```

(from Binding/SigCheck/Service/app/src/main/java/com/commonsware/android/advservice/remotebinding/sig/DownloadService.java)

A `Binder` can find out who is invoking one of its exposed methods via `Binder.getCallingUid()`. This returns the Linux user ID (uid) that the client uses.

Normally, this will be tied to one application ID. However, it is possible for a suite of apps to share a Linux user ID, via the `android:sharedUserId` option in the manifest. Hence, the call to map the user ID to an application ID is `getPackagesForUid()` on `PackageManager`, which returns a *list* of application IDs.

So, the revised `download()` method iterates over those application IDs to see if any of them have the expected signature:

```
  @Override
  public void download(String url) {
    boolean ok=false;

    for (String pkg :
      pm.getPackagesForUid(Binder.getCallingUid())) {
      try {
        String otherHash=
          SignatureUtils.getSignatureHash(ctxt, pkg);

        if (expectedHash.equals(otherHash)) {
          ok=true;
          break;
        }
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
          "Exception finding signature hash", e);
      }
    }
```

```
      if (ok) {
        new DownloadThread(url).start();
      }
      else {
        Log.e(getClass().getSimpleName(),
          "Could not validate client signature");
      }
    }
```

(from Binding/SigCheck/Service/app/src/main/java/com/commonsware/android/advservice/remotebinding/sig/DownloadService.java)

In practice, Android itself will ensure that if there are several application IDs sharing a Linux user ID, they will all be signed by the same signing key.

If and only if we find a signature match do we actually do the download; otherwise, we log an error.

This happens to be a very simple service with a single-method `Binder`. In a more complicated service, where there are several methods exposed by the `Binder`, the signature-check logic could be refactored into a common `private` method that the AIDL-defined `Binder` methods could all use to validate the client.

So, Where Do We Get the Expected Hash From?

Today, there are two main ways you can get the expected hash:

- Since this is really a hash of the public part of the app's signing key, the author of the other app might publish it as part of integration documentation, where the hash is generated via `keytool`
- You might call `getSignatureHash()` from your app and log the results, running it against a known good copy of the other app

Servicing the Service

However, we do not get any result back from the service to know if the download succeeded or failed. That is likely to be rather important information for the user.

In principle, `download()` could return some success-or-failure indication... but then we would have a blocking call. Neither the client nor the service could proceed until the download is completed. That would require the client to manage its own background thread, which is a minor hassle. It also means that the service ties up one of a limited number of "`Binder` threads", which is not a good idea.

Another approach would be to pass some sort of callback object with `download()`, such that the server could run the script asynchronously and invoke the callback on success or failure. This, though, implies that there is some way to have the client export an API to the service.

Fortunately, this is eminently doable, as you will see in this section, and the accompanying samples (`Binding/Callback/Service` and `Binding/Callback/Client`).

Callbacks via AIDL

AIDL does not have any concept of direction. It just knows interfaces, proxies, and stub implementations. In the preceding example, we used AIDL to have the service flesh out the stub implementation and have the client access the service via the AIDL-defined interface. However, there is nothing magic about services implementing interfaces and clients accessing them — it is equally possible to reverse matters and have the client implement something the service uses via an interface.

So, for example, we could create an `IDownloadCallback.aidl` file:

```
package com.commonsware.android.advservice.callbackbinding;

// Declare the interface.
interface IDownloadCallback {
  void onSuccess();
  void onFailure(String msg);
}
```

(from Binding/Callback/Service/app/src/main/aidl/com/commonsware/android/advservice/callbackbinding/IDownloadCallback.aidl)

Then, we can augment `IDownload` itself, to pass an `IDownloadCallback` with `download()`:

```
package com.commonsware.android.advservice.callbackbinding;

import com.commonsware.android.advservice.callbackbinding.IDownloadCallback;

// Declare the interface.
interface IDownload {
  void download(String url, IDownloadCallback cb);
}
```

(from Binding/Callback/Service/app/src/main/aidl/com/commonsware/android/advservice/callbackbinding/IDownload.aidl)

Notice that we need to specifically import `IDownloadCallback`, just like we might import some "regular" Java interface. And, as before, we need to make sure the client and the server are working off of the same AIDL definitions, so these two AIDL files need to be replicated across each project.

But other than that one little twist, this is all that is required, at the AIDL level, to have the client pass a callback object to the service: define the AIDL for the callback and add it as a parameter to some service API call.

Of course, there is a little more work to do on the client and server side to make use of this callback object.

Revising the Client

On the client, we need to implement an `IDownloadCallback`. In `onSuccess()` and `onFailure()` we can do something like raise a `Toast`.

The catch is that we cannot be certain we are being called on the UI thread in our callback object. In fact, it is almost assured that we are not. So, we need to get our work moved over to the main application thread. To do that, this sample uses `runOnUiThread()`:

```
IDownloadCallback.Stub cb=new IDownloadCallback.Stub() {
  @Override
  public void onSuccess() throws RemoteException {
    getActivity().runOnUiThread(new Runnable() {
      @Override
      public void run() {
        Toast.makeText(getActivity(), "Download successful!", Toast.LENGTH_LONG).show();
      }
    });
  }

  @Override
  public void onFailure(final String msg) throws RemoteException {
    getActivity().runOnUiThread(new Runnable() {
      @Override
      public void run() {
        Toast.makeText(getActivity(), msg, Toast.LENGTH_LONG).show();
      }
    });
  }
};
```

(from Binding/Callback/Client/app/src/main/java/com/commonsware/android/advservice/callbackbinding/client/DownloadFragment.java)

And, of course, we need to pass the `IDownloadCallback` object in our `download()` call:

```
@Override
public void onClick(View view) {
  try {
    binding.download(TO_DOWNLOAD, cb);
  }
  catch (RemoteException e) {
    Log.e(getClass().getSimpleName(), "Exception requesting download", e);
    Toast.makeText(getActivity(), e.getMessage(), Toast.LENGTH_LONG).show();
  }
}
```

(from Binding/Callback/Client/app/src/main/java/com/commonsware/android/advservice/callbackbinding/client/DownloadFragment.java)

Revising the Service

The service also needs changing, to use the supplied callback object for the end results of the download.

`DownloadBinder` now receives an `IDownloadCallback` proxy in its `download()` method, which it passes along to the `DownloadThread`:

```
private static class DownloadBinder extends IDownload.Stub {
  @Override
  public void download(String url, IDownloadCallback cb) {
    new DownloadThread(url, cb).start();
  }
}
```

(from Binding/Callback/Service/app/src/main/java/com/commonsware/android/advservice/callbackbinding/svc/DownloadService.java)

Notice that the service's own API just needs the `IDownloadCallback` parameter, which can be passed around and used like any other Java object. The fact that it happens to cause calls to be made synchronously back to the remote client is invisible to the service.

`DownloadThread`, in turn, invokes `onSuccess()` or `onFailure()` as appropriate:

```
private static class DownloadThread extends Thread {
  String url=null;
  IDownloadCallback cb=null;

  DownloadThread(String url, IDownloadCallback cb) {
    this.url=url;
    this.cb=cb;
  }

  @Override
```

```
  public void run() {
    boolean succeeded=false;

    try {
      File root=
          Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

      root.mkdirs();

      File output=new File(root, Uri.parse(url).getLastPathSegment());

      if (output.exists()) {
        output.delete();
      }

      HttpURLConnection c=(HttpURLConnection)new URL(url).openConnection();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;

        while ((len=in.read(buffer)) >= 0) {
          out.write(buffer, 0, len);
        }

        out.flush();
        succeeded=true;
      }
      finally {
        fos.getFD().sync();
        out.close();
        c.disconnect();
      }
    }
    catch (IOException e2) {
      Log.e("DownloadJob", "Exception in download", e2);

      try {
        cb.onFailure(e2.getMessage());
      }
      catch (RemoteException e) {
        Log.e("DownloadJob", "Exception when calling onFailure()", e2);
      }
    }

    if (succeeded) {
      try {
        cb.onSuccess();
      }
      catch (RemoteException e) {
        Log.e("DownloadJob", "Exception when calling onSuccess()", e);
      }
    }
  }
}
```

(from Binding/Callback/Service/app/src/main/java/com/commonsware/android/advservice/callbackbinding/svc/DownloadService.java)

Thinking About Security

Remote services, by definition, are available for anyone to connect to. This may or may not be a good idea.

If the only client of your remote service is some other app of yours, you could protect the service using a custom signature-level permission.

If you anticipate third-party apps communicating with your service, you should strongly consider protecting the service with an ordinary custom permission, so the *user* can vote on whether the communication is allowed.

For local services, the simplest way to secure the service is to not export it, typically by not having an `<intent-filter>` element for the `<service>` in the manifest. Then, your app is the only app that can work with the service.

The "Everlasting Service" Anti-Pattern

One anti-pattern that is all too prevalent in Android is the "everlasting service". Such a service is started via `startService()` and never stops — the component starting it does not stop it and it does not stop itself via `stopSelf()`.

Why is this an anti-pattern?

1. The service takes up memory all of the time. This is bad in its own right if the service is not continuously delivering sufficient value to be worth the memory.
2. Users, fearing services that sap their device's CPU or RAM, may attack the service with so-called "task killer" applications or may terminate the service via the Settings app, thereby defeating your original goal.
3. Android itself, due to user frustration with sloppy developers, will terminate services it deems ill-used, particularly ones that have run for quite some time.

Occasionally, an everlasting service is the right solution. Take a VOIP client, for example. A VOIP client usually needs to hold an open socket with the VOIP server to know about incoming calls. The only way to continuously watch for incoming

calls is to continuously hold open the socket. The only component capable of doing that would be a service, so the service would have to continuously run.

However, in the case of a VOIP client, or a music player, the user is the one specifically requesting the service to run forever. By using `startForeground()`, a service can ensure it will not be stopped due to old age for cases like this.

As a counter-example, imagine an email client. The client wishes to check for new email messages periodically. The right solution for this is the `AlarmManager` pattern described elsewhere in this book. The anti-pattern would have a service running constantly, spending most of its time waiting for the polling period to elapse (e.g., via `Thread.sleep()`). There is no value to the user in taking up RAM to watch the clock tick. Such services should be rewritten to use `AlarmManager`.

Most of the time, though, it appears that services are simply leaked. That is one advantage of using `AlarmManager` and an `IntentService` – it is difficult to leak the service, causing it to run indefinitely. In fact, `IntentService` in general is a great implementation to use whenever you use the command pattern, as it ensures that the service will shut down eventually. If you use a regular service, be sure to shut it down when it is no longer actively delivering value to the user.

Advanced Manifest Tips

If you have been diligent about reading this book (versus having randomly jumped to this chapter), you will already have done a fair number of things with your project's `AndroidManifest.xml` file:

1. Used it to define components, like activities, services, content providers, and manifest-registered broadcast receivers
2. Used it to declare permissions your application requires, or possibly to define permissions that other applications need in order to integrate with your application
3. Used it to define what SDK level, screen sizes, and other device capabilities your application requires

In this chapter, we continue looking at things the manifest offers you, starting with a discussion of controlling where your application gets installed on a device, and wrapping up with a bit of information about activity aliases.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Just Looking For Some Elbow Room

On October 22, 2008, the HTC Dream was released, under the moniker of "T-Mobile G1", as the first production Android device.

Complaints about the lack of available storage space for applications probably started on October 23rd.

The Dream, while a solid first Android device, offered only 70MB of on-board flash for application storage. This storage had to include:

1. The Android application (APK) file
2. Any local files or databases the application created, particularly those deemed unsafe to put on the SD card (e.g., privacy)
3. Extra copies of some portions of the APK file, such as the compiled Dalvik bytecode, which get unpacked on installation for speed of access

It would not take long for a user to fill up 70MB of space, then have to start removing some applications to be able to try others.

Users and developers alike could not quite understand why the Dream had so little space compared to the available iPhone models, and they begged to at least allow applications to install to the SD card, where there would be more room. This, however, was not easy to implement in a secure fashion, and it took until Android 2.2 for the feature to become officially available.

If your app's `android:minSdkVersion` is 11 or higher, you can probably ignore all of this. At that time, what the Android SDK refers to as "internal storage" and "external storage" were moved to be part of one filesystem partition, and so there is no artificial division of space between the two.

But, if you are still supporting Android 2.2 and 2.3, you may wish to consider supporting having your app be installed to, or moved to, external storage.

Configuring Your App to Reside on External Storage

Indicating to Android that your application can reside on the SD card is easy... and necessary, if you want the feature. If you do not tell Android this is allowed, Android will *not* install your application to the SD card, nor allow the user to move the application to the SD card.

All you need to do is add an `android:installLocation` attribute to the root `<manifest>` element of your `AndroidManifest.xml` file. There are three possible values for this attribute:

- `internalOnly`, the default, meaning that the application cannot be installed to the SD card
- `preferExternal`, meaning the application would like to be installed on the SD card
- `auto`, meaning the application can be installed in either location

If you use `preferExternal`, then your application will be initially installed on the SD card in most cases. Android reserves the right to still install your application on internal storage in cases where that makes too much sense, such as there not being an SD card installed at the time.

If you use `auto`, then Android will make the decision as to the installation location, based on a variety of factors. In effect, this means that `auto` and `preferExternal` are functionally very similar – all you are doing with `preferExternal` is giving Android a hint as to your desired installation destination.

Because Android decides where your application is initially installed, and because the user has the option to move your application between the SD card and on-board flash, you cannot assume any given installation spot. The exception is if you choose `internalOnly`, in which case Android will honor your request, at the potential cost of not allowing the installation at all if there is no more room in on-board flash.

For example, here is the manifest from the `SMS/Sender` sample project, profiled in another chapter, showing the use of `preferExternal`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.sender"
  android:installLocation="preferExternal"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>
```

```
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="Sender"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from SMS/Sender/app/src/main/AndroidManifest.xml)

Since this feature only became available in Android 2.2, to support older versions of Android, just have your build tools target API level 8 (e.g., `compileSdkVersion` of 8 or higher in `build.gradle` for Android Studio users) while having your `minSdkVersion` attribute in the manifest state the lowest Android version your application supports overall. Older versions of Android will ignore the `android:installLocation` attribute. So, for example, in the above manifest, the `Sender` application supports API level 4 and above (Android 1.6 and newer), but still can use `android:installLocation="preferExternal"`, because the build tools are targeting API level 8.

What the User Sees

On newer devices, such as those running Android 4.2, the user will see nothing different. That is because internal and external storage share a common pool of space, and therefore there is no advantage in having your application installed to external storage.

However, on, say, Android 2.3, you will see a difference in behavior.

For an application that wound up on external storage, courtesy of your choice of `preferExternal` or `auto`, the user will have an option to move it to the phone's internal storage. This can be done by choosing the application in the Manage Applications list in the Settings application, then clicking the "Move to phone" button.

Conversely, if your application is installed in on-board flash, and it is movable to external storage, they will be given that option with a "Move to SD card" button.

What the Pirate Sees

Ideally, the pirate sees nothing at all.

One of the major concerns with installing applications to the SD card is that the SD card is usually formatted FAT32 (`vfat`), offering no protection from prying eyes. The concern was that pirates could then just pluck the APK file off external storage and distribute it, even for paid apps from the Play Store.

Apparently, they solved this problem.

To quote the Android developer documentation:

> The unique container in which your application is stored is encrypted with a randomly generated key that can be decrypted only by the device that originally installed it. Thus, an application installed on an SD card works for only one device.

Moreover, this "unique container" is not normally mounted when the user mounts external storage on their host machine. The user mounts `/mnt/sdcard`; the "unique container" is `/mnt/asec`.

What Your App Sees… When External Storage is Inaccessible

So far, this has all seemed great for users and developers. Developers need to add a single attribute to the manifest, and Android 2.2+ users gain the flexibility of where the app gets stored.

Alas, there is a problem, and it is a big one: on Android 1.x and 2.x, either the host PC or the device can have access to the SD card, but not both. As a result, if the user makes the SD card available to the host PC, by plugging in the USB cable and mounting the SD card as a drive via a `Notification` or other means, that SD card becomes unavailable for running applications.

So, what happens?

1. First, your application is terminated forcibly, as if your process was being closed due to low memory. Notably, your activities and services will not be

called with `onDestroy()`, and instance state saved via `onSaveInstanceState()` is lost.

2. Second, your application is unhooked from the system. Users will not see your application in the launcher, your `AlarmManager` alarms will be canceled, and so on.
3. When the user makes external storage available to the phone again, your application will be hooked back into the system and will be once again available to the user (for example, your icon will reappear in the launcher)

The upshot: if your application is simply a collection of activities, otherwise not terribly connected to Android, the impact on your application is no different than if the user reboots the phone, kills your process via a so-called "task killer" application, etc. If, however, you are doing more than that, the impacts may be more dramatic.

Perhaps the most dramatic impact, from a user's standpoint, will be if your application implements app widgets. If the user has your app widget on her home screen, that app widget will be removed when the SD card becomes unavailable to the phone. Worse, your app widget cannot be re-added to the home screen until the phone is rebooted (a limitation that hopefully will be lifted sometime after Android 2.2).

The user is warned about this happening, at least in general:

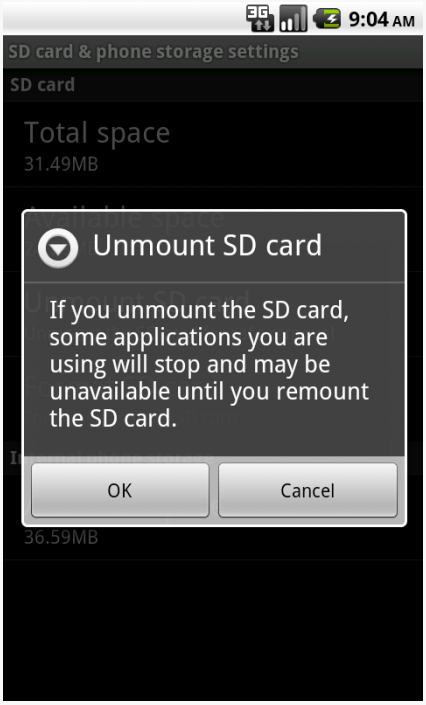


Figure 900: Warning when unmounting the SD card

Two broadcast Intents are sent out related to this:

- `ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE`, when the SD card (and applications installed upon it) become unavailable
- `ACTION_EXTERNAL_APPLICATIONS_AVAILABLE`, when the SD card and its applications return to normal

Note that the documentation is unclear as to whether your own application, that had been on the SD card, can receive `ACTION_EXTERNAL_APPLICATIONS_AVAILABLE` once the SD card is back in action.

Also note that all of these problems hold true for longer if the user physically removes the SD card from the device. If, for example, they replace the card with a different one — such as one with more space — your application will be largely lost. They will see a note in their applications list for your application, but the icon will indicate it is on external storage, and the only thing they can do is uninstall it:

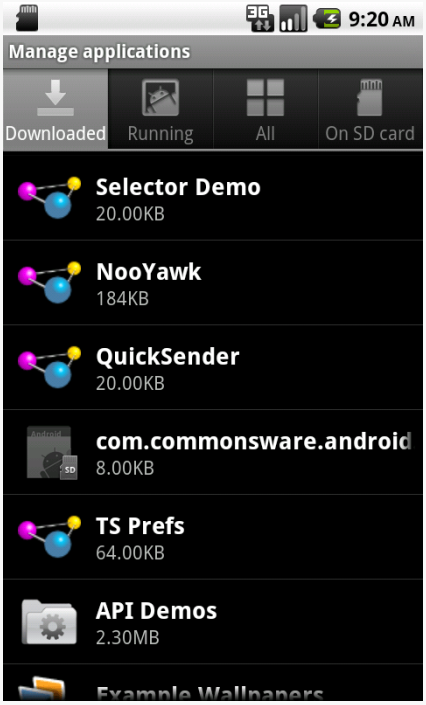


Figure 901: The Manage Applications list, with an application shown from a removed SD card

Choosing Whether to Support External Storage

Given the huge problem from the previous section, the question of whether or not your application should support external storage is far from clear.

As the Android developer documentation states:

> Large games are more commonly the types of applications that should allow installation on external storage, because games don't typically provide additional services when inactive. When external storage becomes unavailable and a game process is killed, there should be no visible effect when the storage becomes available again and the user restarts the game (assuming that the game properly saved its state during the normal Activity lifecycle).

Conversely, if your application implements any of the following features, it may be best to not support external storage:

1. Polling of Web services or other Internet resources via a scheduled alarm

2. Account managers and their corresponding sync adapters, for custom sources of contact data
3. App widgets, as noted in the previous section
4. Device administration extensions
5. Live folders
6. Custom soft keyboards ("input method engines")
7. Live wallpapers
8. Custom search providers

But, as noted earlier, this is not even usually necessary on API Level 11+ devices. Hence, even if your app would otherwise qualify for being installed to external storage, you may not wish to bother. If few devices (Android 2.2 and Android 2.3) might need the capability, it may not be worth the extra testing burden.

Android 6.0 and "Adoption" of Removable Storage

When Android 3.0 did away with the required separate partitions for internal storage and external storage, the `android:installLocation` option fell out of use, as there was no particular value in having the apps on external storage. For single-partition devices — meaning, for most devices — users did not even have the option for moving their apps to external storage.

However, `android:installLocation` is returning to relevance, once again courtesy of removable media.

On Android 6.0+, users with removable storage options, such as micro SD card slots, have the option of "adopting" those as an extension of the device's internal storage. Once done, apps set with `auto` or `preferExternal` for `android:installLocation` can be moved to the removable media. However, there appears to be one key difference: not only is the APK on the removable media, but so is all of that app's portion of internal storage. The removable media is encrypted, so the material copied to the removable media should remain fairly inaccessible.

From the user's standpoint, for low-end devices with minimal on-board flash, they have additional storage space that they can use for apps.

However:

- Removable media tends to be slow, and some cards will be slower than others. For developers, this makes it all that much more important for you to move disk I/O off of the main application thread.

- Removable media tends to be removable. If the user removes the removable media, while your app is installed on that removable media, your app will no longer work.
- All the old rules for apps that allow themselves to be installed on external storage will still hold true. Basically, any app that does periodic work, or will respond to incoming GCM messages, or has an app widget, or is always possibly needed (e.g., custom soft keyboard), should not allow itself to be moved to removable media. If the user *does* eject the media, they will get a permanent `Notification` telling them to put it back:

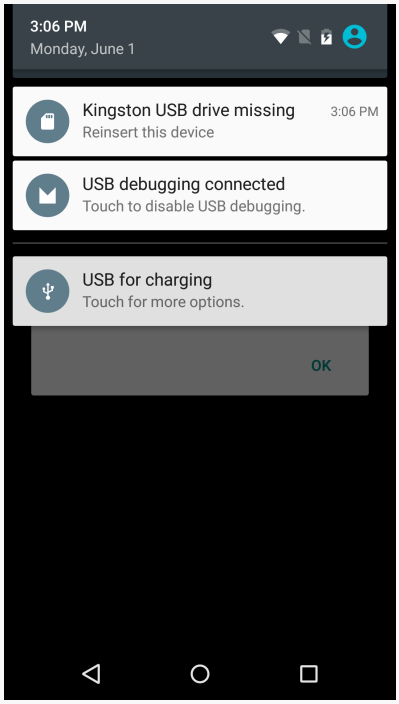


Figure 902: Android 6.0, Ejected Adopted Removable Media Notification

The user does have an "Erase & Format" option that will reformat the removable media and allow it to be permanently removed from the device. It does not appear that this will automatically move any apps back to internal storage. The users would need to move those apps back to internal storage by means of the Apps list in Settings.

Normally, it appears this system will be limited to internal card slots for things like micro SD cards. While USB On-The-Go (OTG) allows Android devices to access thumb drives, those are likely to be accidentally removed by the user (not to mention they usually tie up the charging port). However, for development testing

purposes, you can run the `adb shell sm set-force-adoptable true` command to allow the device to adopt USB OTG drives. Note though that once you do this, the drive is more or less owned by that Android device until you "Erase & Format" it, and you will lose everything on the drive as part of this whole process.

Using an Alias

As was mentioned in the chapter on integration, you can use the `PackageManager` class to enable and disable components in your application. This works at the component level, meaning you can enable and disable activities, services, content providers, and broadcast receivers. It does not support enabling or disabling individual `<intent-filter>` stanzas from a given component, though.

Why might you want to do this?

1. Perhaps you have an activity you want to be available for use, but not necessarily available in the launcher, depending on user configuration or unlocking "pro" features or something
2. Perhaps you want to add browser support for certain MIME types, but only if other third-party applications are not already installed on the device

While you cannot control individual `<intent-filter>` stanzas directly, you can have a similar effect via an activity alias.

An activity alias is another manifest element — `<activity-alias>` – that provides an alternative set of filters or other component settings for an already-defined activity. For example, here is the `AndroidManifest.xml` file from the `Manifest/Alias` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
android:versionName="1.0" package="com.commonsware.android.alias">

  <supports-screens android:largeScreens="true" android:normalScreens="true" android:smallScreens="false"/>
  <application android:icon="@drawable/ic_launcher" android:label="@string/app_name">
    <activity android:label="@string/app_name" android:name="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity-alias android:label="@string/app_name2" android:name="ThisIsTheAlias"
android:targetActivity="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
```

```
      </intent-filter>
    </activity-alias>
  </application>
</manifest>
```

(from Manifest/Alias/app/src/main/AndroidManifest.xml)

Here, we have one `<activity>` element, with an `<intent-filter>` to put the activity in the launcher. We also have an `<activity-alias>` element... which puts a second icon in the launcher for the same activity implementation.

An activity alias can be enabled and disabled independently of its underlying activity. Hence, you can have one activity class have several independent sets of intent filters and can choose which of those sets are enabled at any point in time.

For testing purposes, you can also enable and disable these from the command line. Use the `adb shell pm disable` command to disable a component:

```
adb shell pm disable
com.commonsware.android.alias/com.commonsware.android.alias.ThisIsTheAlias
```

... and the corresponding `adb shell pm enable` command to enable a component:

```
adb shell pm enable
com.commonsware.android.alias/com.commonsware.android.alias.ThisIsTheAlias
```

In each case, you supply the package of the application (`com.commonsware.android.alias`) and the class of the component to enable or disable (`com.commonsware.android.alias.ThisIsTheAlias`), separated by a slash.

Getting Meta (Data)

Sometimes, you may want to put more data in the manifest, associated with your components. You will frequently see this for use with libraries or plugin distribution models, where sharing some configuration data between parties could eliminate a bunch of API code that a reuser might need to implement.

To support this, Android offers a `<meta-data>` element as a child of `<activity>`, `<activity-alias>`, `<receiver>`, or `<service>`. Each `<meta-data>` element has an `android:name` attribute plus an associated value, supplied by either an `android:value` attribute (typically for literals) or an `android:resource` attribute (for references to resources).

Other parties can then get at this information via `PackageManager`. So, for example, the implementer of a plugin could have `<meta-data>` elements indicating details of how the plugin should be used (e.g., desired polling frequency), and the host of the plugin could then get that configuration data without the plugin author having to mess around with implementing some Java API for it.

For example, Roman Nurik's DashClock is a lockscreen app widget designed to serve as a replacement for the clock app widget that ships with many Android 4.2+ devices. Not only does it display the time, but it is a plugin host, allowing third party developers to supply "extensions" that can also display data in the app widget. This way, users can set up a single lockscreen app widget and get at a bunch of useful information.

DashClock's extension API makes use of `<meta-data>` to pass configuration data from the extension to DashClock itself. The implementation of a DashClock extension is a service, and so the extension's `<service>` element will have a batch of `<meta-data>` elements with this configuration data:

```
<service android:name=".ExampleExtension"
     android:icon="@drawable/ic_extension_example"
     android:label="@string/extension_title"

android:permission="com.google.android.apps.dashclock.permission.READ_EXTENSION_DATA">
     <intent-filter>
         <action android:name="com.google.android.apps.dashclock.Extension" />
     </intent-filter>
     <meta-data android:name="protocolVersion" android:value="1" />
     <meta-data android:name="description"
         android:value="@string/extension_description" />
     <!-- A settings activity is optional -->
     <meta-data android:name="settingsActivity"
         android:value=".ExampleSettingsActivity" />
 </service>
```

(sample from the DashClock documentation)

Here, the developer can specify:

- What version of the communications protocol is supported, so DashClock can update its protocol over time yet remain backwards-compatible with older extensions, via the `protocolVersion` entry

- What the description is for the extension, used in DashClock's configuration screens to let the user know what available extensions there are, via the `description` entry
- What activity, if any, does the extension supply that allows the user to configure that extension, that DashClock should provide access to from its own settings activity, via the `settingsActivity` entry

In all three cases, DashClock uses `android:value`. Note that `android:value` *does* support the use of resources — the value of `description` is a reference to the `extension_description` string resource, for example.

To retrieve that metdata, an app can ask for `PackageManager.GET_META_DATA` as a flag on `PackageManager` methods for introspection, like `queryIntentActivities()`. In the case of DashClock, it retrieves all implementations of its plugin by asking Android what *services* have an `<intent-filter>` with an `<action>` of `com.google.android.apps.dashclock.Extension`, via `queryIntentServices()`, asking for `PackageManager` to also supply each service's metadata:

```
List<ResolveInfo> resolveInfos = pm.queryIntentServices(
                new Intent(DashClockExtension.ACTION_EXTENSION),
PackageManager.GET_META_DATA);
```

(from the `ExtensionManager.java` file in the DashClock source code)

Each `ResolveInfo` object that comes back in the list will have a `serviceInfo` field containing details of the service. Because `GET_META_DATA` was passed in as a flag, the `serviceInfo` will have a `Bundle` named `metaData` which will contain the key/value pairs specified by the `<meta-data>` elements. DashClock can then grab that data and use it to populate its own object model:

```
for (ResolveInfo resolveInfo : resolveInfos) {
    ExtensionListing listing = new ExtensionListing();
    listing.componentName = new ComponentName(resolveInfo.serviceInfo.packageName,
            resolveInfo.serviceInfo.name);
    listing.title = resolveInfo.loadLabel(pm).toString();
    Bundle metaData = resolveInfo.serviceInfo.metaData;
    if (metaData != null) {
        listing.protocolVersion = metaData.getInt("protocolVersion");
        listing.description = metaData.getString("description");
        String settingsActivity = metaData.getString("settingsActivity");
        if (!TextUtils.isEmpty(settingsActivity)) {
            listing.settingsActivity = ComponentName.unflattenFromString(
                    resolveInfo.serviceInfo.packageName + "/" + settingsActivity);
```

```
        }
    }
```

(from the `ExtensionManager.java` file in the DashClock source code)

The `<meta-data>` element supports five data types for `android:value`:

- String
- Integer
- Boolean (specified as `true` or `false` in the `android:value` attribute)
- Float

It also supports colors, specified in #AARRGGBB and similar formats, which, according to the documentation, is returned as a string.

In this fashion, extension developers can supply enough information for DashClock to allow the user to see the list of installed extensions, choose which one(s) they want, and configure those (where applicable). Actually getting the *content* to display will need to be done at runtime, in this case via making requests of the service to supply a `ExtensionData` structure with the messages, icon, and so forth to be displayed.

Miscellaneous Integration Tips

This chapter is a collection of other miscellaneous integration and introspection tips and techniques that you might find useful in your Android apps.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Direct Share

The classic means of "sharing" content between apps is via `ACTION_SEND`. You create an `ACTION_SEND` `Intent`, identifying the content to share, and use it with `startActivity()`. The decision of what the candidates are to share with is based solely on the MIME type of the content in question.

Sometimes, sharing of content with another app really means sharing that content with some other person, folder, or finer-grained context within the other app. `ACTION_SEND`, on its own, does not do anything for this. The user chooses the other app, then inside that app chooses the finer-grained context. While `ACTION_SENDTO` supports the sender indicating who to share the content with, that only works for select `Uri` schemes (`mailto` and `smsto`, mostly), and it requires that the sender have a suitable `Uri` to identify the recipient. As a result, few apps support `ACTION_SENDTO`.

Android 6.0 introduced "direct share targets". Now, the recipients of sharing operations can elect to serve up specific share targets, pointing not only to the app but to the finer-grained context within the app. The user will then see these targets listed in the "chooser" window, alongside other standard share targets.

This involves creating a subclass of ChooserTargetService and tying it via some <meta-data> to your activity supporting the ACTION_SEND <intent-filter>. That service will then be called with onGetChooserTargets(), where it is told what activity and <intent-filter> was matched, and the service can return a list of ChooserTarget objects. Those ChooserTarget objects each represent a single direct share target, where the ChooserTarget wraps up a dedicated caption, icon, and PendingIntent for each. Those may be presented to the user in the chooser; if the user chooses one, the PendingIntent is invoked.

The Intents/FauxSenderMNC sample project is a revised version of the FauxSender sample. FauxSender has an implementation of an ACTION_SEND activity, plus a LAUNCHER activity that just uses startActivity() to trigger an ACTION_SEND Intent. FauxSenderMNC augments the original sample with direct-share functionality.

The ChooserTargetService

The bulk of the business logic goes in your subclass of ChooserTargetService, here named CTService:

```
package com.commonsware.android.fsendermnc;

import android.app.PendingIntent;
import android.content.ComponentName;
import android.content.Intent;
import android.content.IntentFilter;
import android.graphics.drawable.Icon;
import android.os.Bundle;
import android.service.chooser.ChooserTarget;
import android.service.chooser.ChooserTargetService;
import java.util.ArrayList;
import java.util.List;

public class CTService extends ChooserTargetService {
  private String titleTemplate;

  @Override
  public void onCreate() {
    super.onCreate();

    titleTemplate=getString(R.string.title_template);
  }

  @Override
  public List<ChooserTarget> onGetChooserTargets(ComponentName sendTarget,
```

```
                                                IntentFilter matchedFilter) {
    ArrayList<ChooserTarget> result=new ArrayList<ChooserTarget>();

    for (int i=1;i<=6;i++) {
      result.add(buildTarget(i));
    }

    return(result);
  }

  private ChooserTarget buildTarget(int targetId) {
    String title=String.format(titleTemplate, targetId);
    int iconId=getResources().getIdentifier("ic_share" + targetId,
        "drawable", getPackageName());
    Icon icon=Icon.createWithResource(this, iconId);
    float score=1.0f-((float)targetId/40);
    ComponentName cn=new ComponentName(this, FauxSender.class);
    Bundle extras=new Bundle();

    extras.putInt(FauxSender.EXTRA_TARGET_ID, targetId);

    return(new ChooserTarget(title, icon, score, cn, extras));
  }
}
```

(from Intents/FauxSenderMNC/app/src/main/java/com/commonsware/android/fsendermnc/CTService.java)

You are welcome to override the `onCreate()` and `onDestroy()` lifecycle methods in your `ChooserTargetService` if you want, though it is not required. Here, we override `onCreate()` just to grab a string resource value that will be used as a template, stashing it in a data member.

The one method that you have to implement is `onGetChooserTargets()`. This will be called when direct-share is triggered, as directed by some manifest entries that we will examine in a bit. Your job is to return a `List` of `ChooserTarget` objects that represent specific ways to share the content into your app, such as sharing to particular contacts or folders or something.

Note that whatever you return from `onGetChooserTargets()` is included *along with your regular `ACTION_SEND` activity itself*. Hence, you only want to return `ChooserTarget` objects that improve the user flow beyond your base `ACTION_SEND` activity — you do not need to have a `ChooserTarget` that simply replicates what the user would get from the `ACTION_SEND` activity itself.

In this case, `onGetChooserTargets()` returns a six-element `ArrayList` of `ChooserTarget` objects, each built using a private `buildTarget()` method.

A `ChooserTarget` is a simple wrapper around five pieces of data:

- A `String` to use as a caption for your direct-share icon
- An `Icon` that represents the icon itself
- A `float` "score" that represents the relative importance of this direct-share target over any others that you return, where `1.0f` means "the user is really going to like this one", `0.0f` means "the user could conceivably want this, but probably not", and values in between 0 and 1 represent shades of gray in the realm of importance
- A `ComponentName` identifying either an activity in your app (the typical answer) or an exported activity in another app (rather unusual)
- A `Bundle` of extras to go into the `Intent` that the framework will create, using that `ComponentName`, to trigger the activity in question

Note that the `ComponentName` does *not* have to start the same activity that is your `ACTION_SEND` activity. In this sample, it happens to use the same activity. But that is not a requirement, and frequently you will use some other activity. For example, if your normal `ACTION_SEND` flow would first have the user choose a folder, then provide additional information about the shared item (e.g., confirm the title, add tags), if you create direct-share targets that specify particular folders, you would want to bypass the folder-selection step in your own UI. If the `ACTION_SEND` activity implements the folder-selection logic and forwarded the user along to some other activity to handle the rest, your `ChooserTarget` `ComponentName` objects might just drive straight to the second activity, skipping the folder-selection UI.

Also note that you may be creating several `ChooserTarget` objects, probably having each pointing to the same activity. You will need to ensure that the extras `Bundle` contains what you need to distinguish one request from the next. However, do not put custom `Parcelable` objects in this `Bundle`, as Android will attempt to un-parcel them as part of its work, and it will fail to do so since Android does not have your custom `Parcelable` class.

An `Icon` is a new construct in Android 6.0, serving as a wrapper around multiple possible image sources. You can create an `Icon` from a drawable resource (as the sample app does), from a `Bitmap`, from a `byte` array representing PNG or JPEG data, from a file path pointing to a PNG or JPEG file, or from a `Uri` to a `ContentProvider` pointing to an image.

The Manifest Entries

Your `ChooserTargetService` will have a typical `<service>` manifest entry, with two special bits:

- An `android:permission="android.permission.BIND_CHOOSER_TARGET_SERVICE"` attribute, to limit access to your service to the framework, rather than being spoofed by other clients, and
- An `<intent-filter>` for the `android.service.chooser.ChooserTargetService` action

```
    <service
      android:name=".CTService"
      android:permission="android.permission.BIND_CHOOSER_TARGET_SERVICE">
      <intent-filter>
        <action android:name="android.service.chooser.ChooserTargetService"/>
      </intent-filter>
    </service>
```

(from Intents/FauxSenderMNC/app/src/main/AndroidManifest.xml)

Your `ACTION_SEND` activity will have its normal `<activity>` element, with just one change: a `<meta-data>` element pointing to your `ChooserTargetService`:

```
    <activity
      android:name="FauxSender"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
      <meta-data
        android:name="android.service.chooser.chooser_target_service"
        android:value=".CTService"/>
    </activity>
```

(from Intents/FauxSenderMNC/app/src/main/AndroidManifest.xml)

It is possible that your app has multiple `ACTION_SEND` activities. In that case, each could have its own `ChooserTargetService`. However, you could elect to have all of

your ACTION_SEND activities route to the same ChooserTargetService if you prefer. onGetChooserTargets() is passed two parameters to help identify where the direct-share request is coming from:

- the ComponentName of the ACTION_SEND activity that was tied to your service, and
- the IntentFilter that triggered that activity in the first place, so you can determine things like the MIME type of the to-be-shared content

Note that you are *not* given the content itself, in the form of the Intent that will eventually be delivered to your ACTION_SEND activity or to your direct-share target via its ComponentName. This is for privacy reasons; otherwise, an app could ask to share anything and be able to peek at anything the user tried sharing with any app.

The Results

The FauxSender activity — the one handling the ACTION_SEND Intent and the direct-share Intent — now looks for the EXTRA_TARGET_ID that the CTService put in its Intent and includes it in the Toast:

```
package com.commonsware.android.fsendermnc;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class FauxSender extends Activity {
  public static final String EXTRA_TARGET_ID="targetId";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    String epilogue="";

    super.onCreate(savedInstanceState);

    int targetId=getIntent().getIntExtra(EXTRA_TARGET_ID, -1);

    if (targetId>0) {
      epilogue=" for target ID #"+targetId;
    }

    String msg=getIntent().getStringExtra(Intent.EXTRA_TEXT);
```

```
    if (TextUtils.isEmpty(msg)) {
      msg=getIntent().getStringExtra(Intent.EXTRA_SUBJECT);
    }

    if (TextUtils.isEmpty(msg)) {
      msg=getString(R.string.no_message_supplied);
    }

    Toast.makeText(this, msg+epilogue, Toast.LENGTH_LONG).show();

    finish();
  }
}
```

(from Intents/FauxSenderMNC/app/src/main/java/com/commonsware/android/fsendermnc/FauxSender.java)

If you run the sample app from Android Studio, the launcher activity will trigger an `ACTION_SEND` of some text. That, in turn, will bring up the chooser panel... but on an Android 6.0 device, that panel will start off with our six direct-share targets:

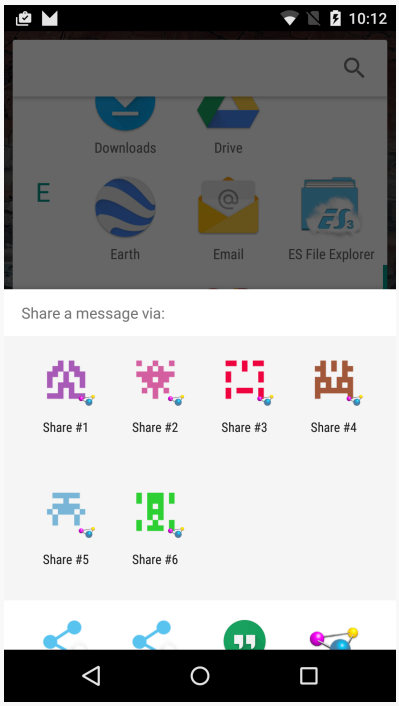


Figure 903: Chooser, Showing Direct-Share Targets

Expanding the panel shows that our original `ACTION_SEND` activity is also there, after the direct-share targets:

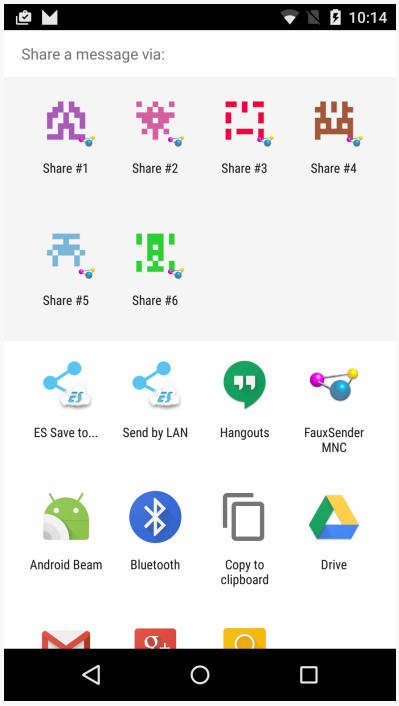


Figure 904: Chooser, Showing More Share Targets

If the user taps on the regular `ACTION_SEND` activity icon, the sample works as it did originally, showing a `Toast` with the text supplied by the launcher activity. If, however, the user taps on one of the direct-share targets, the `Toast` also shows which target was chosen:

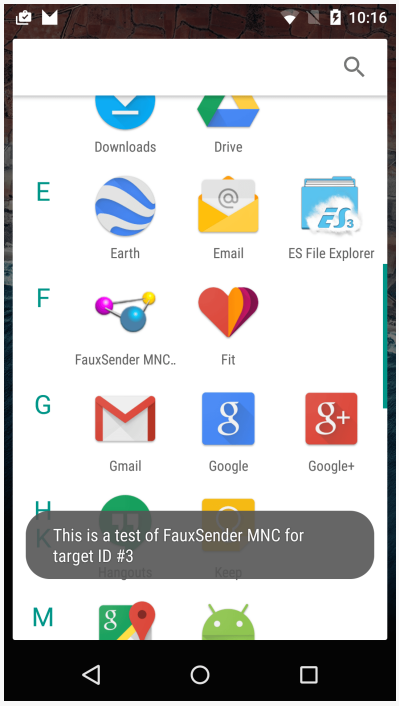


Figure 905: Toast from a Direct-Share Target

Now, our `Bundle` for the direct-share target did not include the shared text, because we did not have it. Instead, the regular `ACTION_SEND` extras are merged in with our own extras, so our activity gets all of the relevant extras.

But… I Got Nothin'!

If you do not have any direct-share targets for a particular request, returning an empty list is perfectly fine.

If you know in advance that you will not have any direct-share targets — for example, the user has not really worked with your app yet after installation — you can disable the service (`android:enabled="false"`). Even though the `<meta-data>` will point to the service, the framework seems to detect the disabled service and continues on unabated.

Even if you elect to leave the service enabled at the outset for Android 6.0, you should consider disabling the service for earlier versions of Android, since it is useless on those devices. You could do this using boolean resources:

- Have a `res/values/bools.xml` file with a `bool` resource (e.g., `offer_direct_share`) set to `false`
- Have a `res/values-v23/bools.xml` file redefining that resource to `true`
- Have `android:enabled="@bool/offer_direct_share"` on your service, to have it be enabled only on Android 6.0 and higher

Best Practices

At the moment, it appears that Android 6.0 is limiting the number of share targets, only showing 8 of them. If you provide more than 8, Android will choose the ones with the highest score.

Since returning the list of direct-share targets should be involving IPC, there may be capacity limitations, for the number and size of the direct-share targets. Do not be surprised if you get a "FAILED BINDER TRANSACTION" exception if your roster of direct-share targets exceeds 1MB.

Hence, between those two limitations, you will want to constrain how many share targets you try returning from your `ChooserTargetService`.

As with other places in Android 5.0+ (e.g., large icons in notifications), your app's icon will be applied as a badge over the icons that you use for direct-share targets. Make sure that your app's icon will work both as a launcher icon and as a direct-share target badge.

Take the Shortcut

Another way to integrate with Android is to offer custom shortcuts. Shortcuts are available from the home screen. Whereas app widgets allow you to draw on the home screen, shortcuts allow you to wrap a custom `Intent` with an icon and caption and put that on the home screen. You can use this to drive users not just to your application's "front door", like the launcher icon, but to some specific capability within your application, like a bookmark.

In our case, in the `Introspection/QuickSender` sample project, we will allow users to create shortcuts that use `ACTION_SEND` to send a pre-defined message, either to a specific address or anywhere, as we have seen before in this chapter.

Once again, the key is in the intent filter.

Registering a Shortcut Provider

Here is the manifest for `QuickSender`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.commonsware.android.qsender" android:versionCode="1" android:versionName="1.0">

  <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="15"/>
  <application android:icon="@drawable/ic_launcher" android:label="@string/app_name">
    <activity android:name="QuickSender" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.CREATE_SHORTCUT"/>
        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

(from Introspection/QuickSender/app/src/main/AndroidManifest.xml)

Our single activity does not implement a traditional launcher `<intent-filter>`. Rather, it has one that watches for a `CREATE_SHORTCUT` action. This does two things:

- It means that our activity will show up in the list of possible shortcuts a user can configure
- It means this activity will be the recipient of a `CREATE_SHORTCUT` Intent if the user chooses this application from the shortcuts list

Implementing a Shortcut Provider

The job of a shortcut-providing activity is to:

1. Create an `Intent` that will be what the shortcut launches
2. Return that `Intent` and other data to the activity that started the shortcut provider
3. Finally, `finish()`, so the caller gets control

You can see all of that in the `QuickSender` implementation:

```
package com.commonsware.android.qsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
```

```
import android.widget.TextView;

public class QuickSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void save(View v) {
    Intent shortcut=new Intent(Intent.ACTION_SEND);
    TextView addr=(TextView)findViewById(R.id.addr);
    TextView subject=(TextView)findViewById(R.id.subject);
    TextView body=(TextView)findViewById(R.id.body);
    TextView name=(TextView)findViewById(R.id.name);

    if (!TextUtils.isEmpty(addr.getText())) {
      shortcut.putExtra(Intent.EXTRA_EMAIL,
                        new String[] { addr.getText().toString() });
    }

    if (!TextUtils.isEmpty(subject.getText())) {
      shortcut.putExtra(Intent.EXTRA_SUBJECT, subject.getText()
                                                     .toString());
    }

    if (!TextUtils.isEmpty(body.getText())) {
      shortcut.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
    }

    shortcut.setType("text/plain");

    Intent result=new Intent();

    result.putExtra(Intent.EXTRA_SHORTCUT_INTENT, shortcut);
    result.putExtra(Intent.EXTRA_SHORTCUT_NAME, name.getText()
                                                    .toString());
    result.putExtra(Intent.EXTRA_SHORTCUT_ICON_RESOURCE,
                    Intent.ShortcutIconResource.fromContext(this,
                                                            R.drawable.icon));

    setResult(RESULT_OK, result);
    finish();
  }
}
```

(from Introspection/QuickSender/app/src/main/java/com/commonsware/android/qsender/QuickSender.java)

The `shortcut Intent` is the one that will be launched when the user taps the shortcut icon on the home screen. The `result Intent` packages up shortcut plus the icon and caption, where the icon is converted into an `Intent.ShortcutIconResource` object. That `result Intent` is then used with the `setResult()` call, to pass that back to whatever called `startActivityForResult()` to open up `QuickSender`. Then, we `finish()`.

At this point, all the information about the shortcut is in the hands of Android (or, more accurately, the home screen application), which can add the icon to the home screen.

Using the Shortcuts

Exactly how `CREATE_SHORTCUT` implementations like this are handled depends on the home screen implementation. Some might not offer them at all. Other home screens might have dedicated options for shortcuts.

The Nexus series devices, running Android 6.0, lump `CREATE_SHORTCUT` implementations in with the app widgets. You can add one to your home screen by long-tapping on the home screen, choosing “Widgets”, and scrolling down to the shortcut that you want:

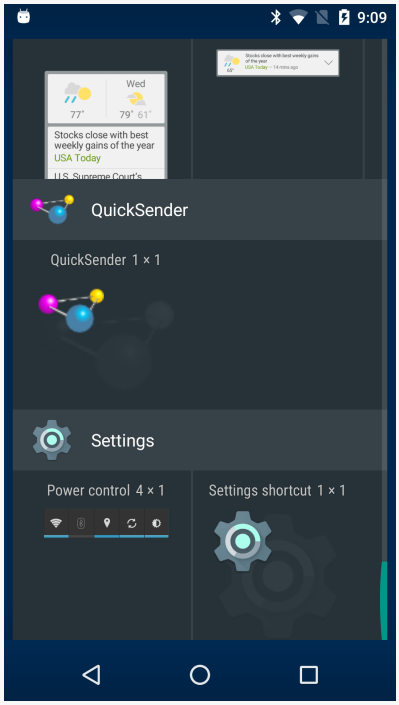


Figure 906: Android 6.0, Widgets List, Showing Sample App

Tap-and-hold on the "widget", and you will be able to place it on the screen. Once that is done, our activity will appear, with the form to define what to send:

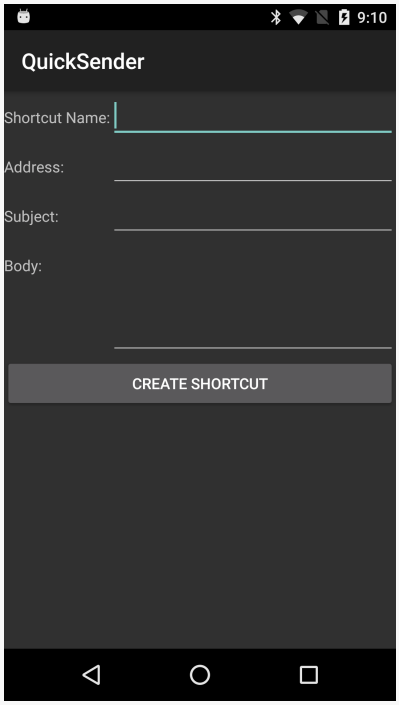


Figure 907: QuickSender Configuration Activity

Fill in the name, either the subject or body, and optionally the address. Then, click the Create Shortcut button, and you will find your shortcut sitting on your home screen, with your chosen shortcut name as the label:

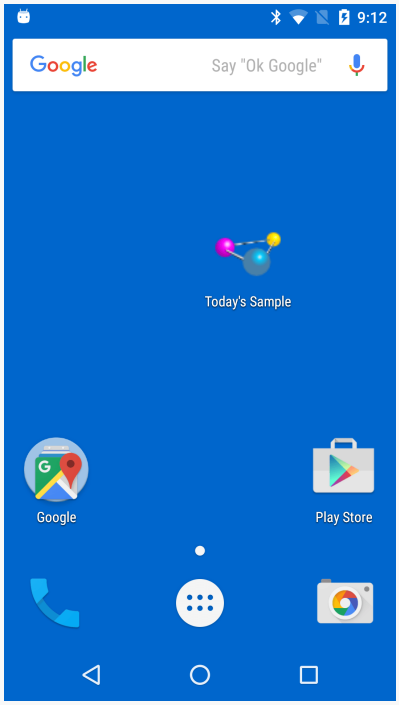


Figure 908: Home Screen, Showing QuickSender-Defined Shortcut

If you launch that shortcut, and if there is more than one application on the device set up to handle `ACTION_SEND`, Android will bring up a special chooser, to allow you to not only pick how to send the message, but optionally make that method the default for all future requests:

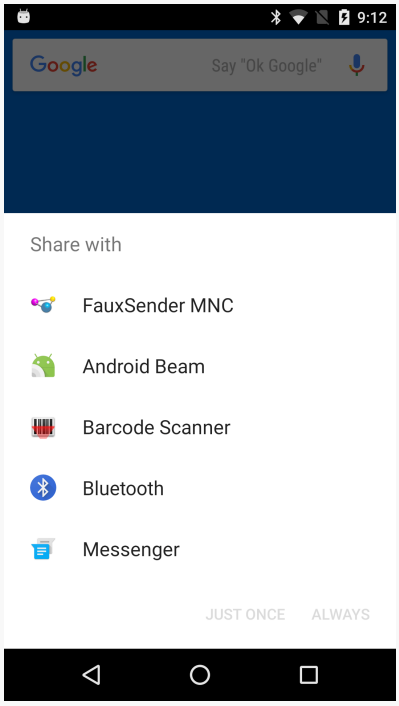


Figure 909: ACTION_SEND Request, As Triggered by Shortcut

Depending on what you choose, of course, will dictate how the message actually gets sent.

Homing Beacons for Intents

If you are encountering problems with `Intent` resolution — you create an `Intent` for something and try starting an `Activity` or `Service` with it, and it does not work — you can add the `FLAG_DEBUG_LOG_RESOLUTION` flag to the `Intent`. This will dump information to Logcat about how the `Intent` resolution occurred, so you can better diagnose what might be going wrong.

Integrating with Text Selection

On Android 6.0+, if you highlight text, you will see a new floating action mode, where cut, copy, and paste operations reside:

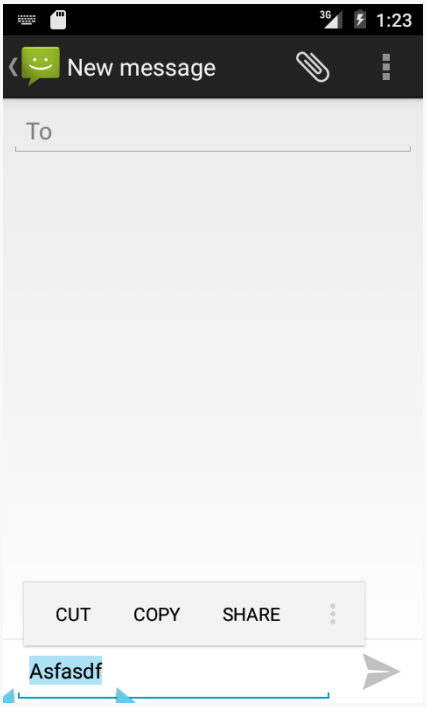


Figure 910: Floating Action Mode

If you tap that overflow indicator on the action mode, a fly-out menu will appear... one that contains arbitrary apps, in addition to system-supplied options:

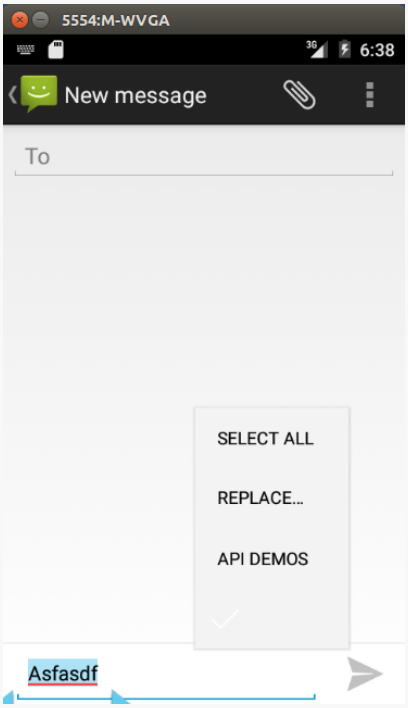


Figure 911: Floating Action Mode, Showing Overflow with Custom Apps

In this case, the Android 6.0 "API Demos" app appears as an option. Choosing it pops up an activity *that has access to the highlighted text from the preceding activity*:

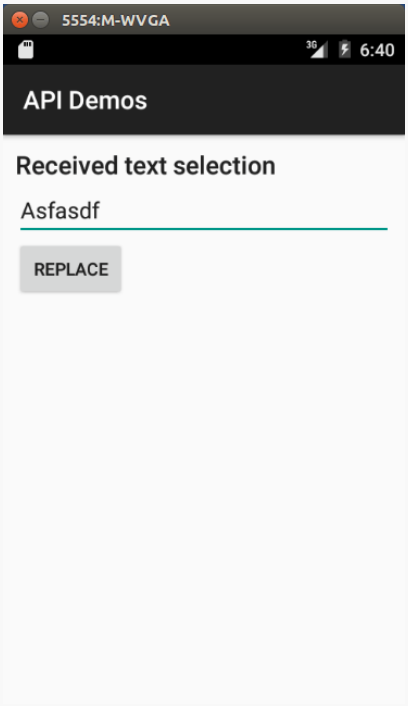


Figure 912: API Demos Application, Showing Text Selection

Replacing the value in the field and clicking the button puts your replacement text in as a replacement for whatever you had highlighted.

This is accomplished via the `ACTION_PROCESS_TEXT` `Intent` action. Apps can advertise activities that support this action, and they will be added (sometimes) to the floating action mode. Apps that have `EditText` widgets just automatically get these options in the floating action mode, with no additional required code.

NOTE: `ACTION_PROCESS_TEXT` support is broken on Android 8.0 and 8.1, and so you may wish to reconsider whether adding `ACTION_PROCESS_TEXT` is worthwhile.

Supporting ACTION_PROCESS_TEXT

Your app can offer an `ACTION_PROCESS_TEXT` activity, in which case you will appear in Android 6.0+ text-selection floating action modes. This is illustrated in the `Introspection/ProcessText` sample application.

The Manifest

To be visible to these text-selection action modes, you need an activity with an `<intent-filter>` calling for `ACTION_PROCESS_TEXT` and a MIME type of `text/plain`:

```
    <activity
      android:name="MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.PROCESS_TEXT"/>
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
      </intent-filter>
    </activity>
```

(from Introspection/ProcessText/app/src/main/AndroidManifest.xml)

Exactly which MIME types are supported is not documented. At the time of this writing, the only examples showed `text/plain`. It is possible that other formats (e.g., `text/html`) might also be supported.

The Extras

You will get one of two extras attached to the `ACTION_PROCESS_TEXT` `Intent`:

- `EXTRA_PROCESS_TEXT` is the text to be processed, and also indicates that you can supply replacement text, if you wish
- `EXTRA_PROCESS_TEXT_READONLY` will be set if `EXTRA_PROCESS_TEXT` is not, and provides the text to be processed and an indication that you cannot supply replacement text

It is up to you to check for those string extras, grab the right value, and do something useful with it.

In the sample app, in `onCreate()` of the `MainActivity`, if we are starting fresh (i.e., there is no `QuestionsFragment` already), we get the search string and provide it to `QuestionsFragment` via a `newInstance()` factory method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
    String search=null;

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M) {
      if (Intent.ACTION_PROCESS_TEXT.equals(getIntent().getAction())) {
        search=getIntent().getStringExtra(Intent.EXTRA_PROCESS_TEXT);

        if (search==null) {
          search=getIntent()
            .getStringExtra(Intent.EXTRA_PROCESS_TEXT_READONLY);
        }
      }
    }

    getFragmentManager()
      .beginTransaction()
      .add(android.R.id.content,
        QuestionsFragment.newInstance(search))
      .commit();
  }
}
```

(from Introspection/ProcessText/app/src/main/java/com/commonsware/android/processtext/MainActivity.java)

`QuestionsFragment`, in turn, stuffs that value into the arguments `Bundle` in `newInstance()`:

```
static QuestionsFragment newInstance(String search) {
  QuestionsFragment result=new QuestionsFragment();
  Bundle args=new Bundle();

  args.putString(ARG_SEARCH, search);
  result.setArguments(args);

  return(result);
}
```

(from Introspection/ProcessText/app/src/main/java/com/commonsware/android/processtext/QuestionsFragment.java)

An expanded version of `StackOverflowInterface` offers not only the original `questions()` method, but also a `search()` method, the latter of which searches Stack Overflow for questions in the `android` tag that have a search term in the title:

```
package com.commonsware.android.processtext;

import retrofit.Callback;
import retrofit.http.GET;
import retrofit.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  void questions(@Query("tagged") String tags, Callback<SOQuestions> cb);
  @GET("/2.2/questions?order=desc&sort=creation&site=stackoverflow&tagged=android")
  void search(@Query("intitle") String search, Callback<SOQuestions> cb);
}
```

(from Introspection/ProcessText/app/src/main/java/com/commonsware/android/processtext/StackOverflowInterface.java)

`onCreateView()` in `QuestionsFragment` then calls either `questions()` or `search()`, depending on whether or not we have a search string from `ACTION_PROCESS_TEXT`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
        super.onCreateView(inflater, container, savedInstanceState);

    setRetainInstance(true);

    RestAdapter restAdapter=
        new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com")
                                 .build();
    StackOverflowInterface so=
        restAdapter.create(StackOverflowInterface.class);
    String search=getArguments().getString(ARG_SEARCH);

    if (search==null) {
      so.questions("android", this);
    }
    else {
      so.search(search, this);
    }

    return(result);
  }
```

(from Introspection/ProcessText/app/src/main/java/com/commonsware/android/processtext/QuestionsFragment.java)

The Results (If Any)

If you got a value for `EXTRA_PROCESS_TEXT` and you wish to return a replacement string, you need to create an `Intent` with your own `EXTRA_PROCESS_TEXT` value that is the replacement text, then use that `Intent` with `setResult()`. `MainActivity` does this when the user taps on a list item in `QuestionsFragment`:

```
@Override
public void onQuestion(Item question) {
  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.M &&
    Intent.ACTION_PROCESS_TEXT.equals(getIntent().getAction()) &&
    getIntent().getStringExtra(Intent.EXTRA_PROCESS_TEXT)!=null) {
    setResult(Activity.RESULT_OK,
      new Intent().putExtra(Intent.EXTRA_PROCESS_TEXT, question.link));
    finish();
  }
  else {
    startActivity(new Intent(Intent.ACTION_VIEW,
      Uri.parse(question.link)));
  }
```

(from Introspection/ProcessText/app/src/main/java/com/commonsware/android/processtext/MainActivity.java)

If the activity was started due to a replaceable bit of text to be processed, we return the URL to the question the user tapped on. In all other cases, we just start up some browser or other app to view that URL.

If you install this app on an Android 6.0+ device, then run some other app that has an `EditText`, type in some term in portrait mode, highlight it, and choose "PROCESS TEXT DEMO" from the floating action mode, you will be presented with a list of Stack Overflow questions in the `android` tag that refer to your search term in the title. If you tap on one, your search term will be replaced in the `EditText` widget by the URL of the question.

Limitations of ACTION_PROCESS_TEXT

Alas, `ACTION_PROCESS_TEXT` is "not all unicorns and rainbows". There are a few issues that you will need to take into account.

Security

There is no documented `android:permission` attribute to place on the `<activity>` that is offering `ACTION_PROCESS_TEXT`, to limit callers. Ideally, we could limit

invocations of ACTION_PROCESS_TEXT only to the firmware itself. As it stands, any app can call startActivity() (or, worse, startActivityForResult()) for your ACTION_PROCESS_TEXT activity and have your code process the text (with user intervention). Please be sure that if you return data via EXTRA_PROCESS_TEXT that the data not include any private information or anything that needs to be secured.

With luck, this will be improved in a future version of Android.

Supporting ACTION_PROCESS_TEXT in Custom Views

TextView and its subclasses are already capable of offering the user ACTION_PROCESS_TEXT options. However, you may have custom View classes that have the notion of text selection, but where you are rendering the available actions to take upon that selection yourself. In that case, you will need to do the reverse: find the implementers of ACTION_PROCESS_TEXT and add them to your UI.

To do this:

- Create an Intent for ACTION_PROCESS_TEXT and a MIME type of text/plain
- Use queryIntentActivities() on PackageManager to find out the activities that handle that Intent structure
- Organize the results, such as sorting them alphabetically by label using ResolveInfo.DisplayNameComparator
- Create Intent objects for each resolved activity, also with ACTION_PROCESS_TEXT and text/plain, but also with EXTRA_PROCESS_TEXT *or* EXTRA_PROCESS_TEXT_READONLY filled in with your selection, and also call setClassName() to provide the package name and activity class name to make the Intent explicit
- Add appropriate elements to your UI for each of those Intent objects
- If the user chooses one, call startActivity() (for EXTRA_PROCESS_TEXT_READONLY) or startActivityForResult() (for EXTRA_PROCESS_TEXT) to invoke the other activity
- In the case of EXTRA_PROCESS_TEXT, watch for your result in onActivityResult() and use the replacement text supplied in the result Intent and its EXTRA_PROCESS_TEXT string extra

The Android Developers Blog has a post that provides some code for this, assuming that you want to put items in an action bar or action mode for the various resolved activities.

Blocking ACTION_PROCESS_TEXT

There will be cases where you do not want `ACTION_PROCESS_TEXT` to be offered to your users. For example, perhaps the text contains sensitive information that should not be passed outside of your app.

The best solution, particularly for a `TextView`, is to mark the text as not being selectable. This is accomplished via `android:textIsSelectable="false"` in a layout file, or via `setTextIsSelectable(false)` in Java. `false` is the default value for `TextView`.

However, for an `EditText` widget, `true` is the default is-selectable state, and you cannot seem to override that with `setTextIsSelectable(false)`.

There is no officially supported option for handling this case, though perhaps there will be one in the future.

One unsupported hack of a workaround relies upon the fact that `EditText` blocks the floating action mode for password fields. In the source code to `EditText`, `TextView`, and related classes, this is handled by seeing if the `TransformationMethod` associated with the widget is `PasswordTransformationMethod`. A `TransformationMethod` is responsible for on-the-fly adjustments between what the user types and what the user sees, such as `PasswordTransformationMethod` replacing typed-in characters with dots.

Making your `EditText` widget use `PasswordTransformationMethod` itself is fine for actual password fields. But suppose you have an `EditText` whose contents should be kept private but should not have the input-shrouding effect of `PasswordTransformationMethod`. To offer this, you would need to create a *subclass* of `PasswordTransformationMethod` (so the block-the-floating-action-mode logic works) that does not actually transform the text (to block the changes that `PasswordTransformationMethod` would ordinarily apply).

A proof-of-concept implementation of this can be found in the `Introspection/ProcessTextBlocker` sample application. This is a clone of the `FilesEditor` sample app from the chapter on files, with one change: the use of `DummyTransformationMethod`:

```
private static class DummyTransformationMethod
  extends PasswordTransformationMethod {
  @Override
```

```
  public CharSequence getTransformation(CharSequence source,
                                        View view) {
    return(source);
  }

  @Override
  public void onTextChanged(CharSequence s, int start,
                            int before, int count) {
    // no-op
  }

  @Override
  public void onFocusChanged(View view,
                             CharSequence sourceText,
                             boolean focused, int direction,
                             Rect previouslyFocusedRect) {
    // no-op
  }

  @Override
  public void afterTextChanged(Editable s) {
    // no-op
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start,
                                int count, int after) {
    // no-op
  }
}
```

(from Introspection/ProcessTextBlocker/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

This is a do-nothing `TransformationMethod`. Ordinarily, this would be completely useless. However, it inherits from `PasswordTransformationMethod`, which is what we need to block the floating action mode.

In `onCreateView()` of the `EditorFragment`, we apply a `DummyTransformationMethod` via `setTransformationMethod()`:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);

  editor=(EditText)result.findViewById(R.id.editor);
```

```
    editor.setTransformationMethod(new DummyTransformationMethod());

    return(result);
  }
```

(from Introspection/ProcessTextBlocker/app/src/main/java/com/commonsware/android/fileseditor/EditorFragment.java)

However, this approach has limitations:

- It only works in portrait, not landscape, for unclear reasons. Since `ACTION_PROCESS_TEXT` *also* only works in portrait, not landscape, we still succeed in blocking `ACTION_PROCESS_TEXT` options.
- It blocks the entire floating action mode (in portrait), clobbering the existing cut/copy/paste/select-all options that might ordinarily be there.
- Since it is tied to internal implementation (that the floating action mode is suppressed when using an `instanceof` a `PasswordTransformationMethod`), not only is this subject to change across Android versions, but also it is subject to change based on device manufacturer or custom ROM tweaks to the Android source code.

Quick Settings and TileService

Android 5.0 added "quick settings" tiles to the notification shade. Android 7.0 allows developers to define their own tiles.

However, to paraphrase Marvel Comics, "with great power comes great need to actually think this through".

Such tiles are only needed in cases where:

- You are doing background work that the user might need to configure
- You do not have a `Notification` tied to that background work, such as through a foreground service (as, in that case, they can and should interact with the `Notification`)

So, if your work is driven by things like `AlarmManager`, `JobScheduler`, or GCM, having your own custom tile may be reasonable. Similarly, if your app is serving as a bridge to some external hardware, via USB, Bluetooth, or other protocols, offering a tile may be useful.

However, Google seems concerned about the scenarios where this gets used:

> Quick Settings tiles are reserved for controls or actions that are either urgently required or frequently used, and should not be used as shortcuts to launching an app.

Hence, be *very* judicious about where you use this capability, lest Google decide to start banning apps for having tiles that do not meet their intended use cases.

Assuming that you feel that your use case is valid, you can implement a `TileService` and publish a `Tile`. The `Tile` contains the icon and caption that will be shown to the user. You can find out when the tile is tapped (e.g., to start an activity to manage whatever the tile is showing) and arrange to update the tile if needed (e.g., reflecting changes in the state of the external hardware or your connection to it).

To see this in action, let's examine part of the `Introspection/SAWMonitor` sample project. This app monitors for installations of apps and updates of apps. If an app is installed or updated, and that app has requested the `SYSTEM_ALERT_WINDOW` permission (and the app is not on a user-maintained whitelist), the app raises a `Notification`. On Android 7.1 and older devices, `SAWMonitor` monitors for app installs and updates via a manifest-registered receiver; on Android 8.0+, it uses a `JobService` to poll for changes periodically.

Other aspects of the `SAWMonitor` sample app can be found in the chapter on advanced preferences.

On Android 7.0+ devices, `SAWMonitor` offers a notification shade tile via a `TileService`. This `TileService` enables and disables the monitoring. The user can also accomplish this through the `MainActivity` and its "enabled" `SwitchPreference`. However, since this app does not need an always-on `Notification`, offering the tile to the user gives the user flexibility to use the tile for rapidly enabling or disabling the monitor.

The Manifest Entry

Your app's manifest will need to have a `<service>` element pointing to your `TileService` subclass. That `<service>` element has some specific requirements, if you want your tile to work:

```
    <service
      android:name=".ToggleTileService"
      android:icon="@drawable/ic_new_releases_24dp"
      android:label="@string/app_name"
      android:enabled="@bool/is_n"
```

```
      android:permission="android.permission.BIND_QUICK_SETTINGS_TILE">
      <intent-filter>
        <action android:name="android.service.quicksettings.action.QS_TILE" />
      </intent-filter>
      <meta-data
        android:name="android.service.quicksettings.ACTIVE_TILE"
        android:value="false" />
    </service>
```

(from Introspection/SAWMonitor/app/src/main/AndroidManifest.xml)

Specifically:

- It needs the `android.service.quicksettings.action.QS_TILE` `<intent-filter>`, so Android knows that you are publishing a tile
- It needs `android:permission="android.permission.BIND_QUICK_SETTINGS_TILE"`, so that only the system can bind to your service
- It needs the `android:label` and `android:icon` attributes, pointing to resources that will make up the default content of your tile
- Optionally, it can have the `<meta-data>` element, with a name of `android.service.quicksettings.ACTIVE_TILE`, and a boolean value indicating whether this tile is an "active tile" or not (more on this later)

Ordinarily, you could skip the label and icon, inheriting the values from the `<application>`. In this case, the tile's icon needs to more closely resemble a modern `Notification` icon: an alpha channel mask, not a full-color icon. Hence, most likely you would be overriding these attributes here anyway.

Active and Passive Tiles

Your tile can be considered "active" or "passive". The default is "passive", where you will be told to update the tile's contents when the user slides open the notification shade. For most situations, this will be fine. But, it may be that you need to show real-time updates while that shade is open. In that case, you need an "active" update model. To opt into that, your `<service>` also needs the following child element:

```
<meta-data
  android:name="android.service.quicksettings.ACTIVE_TILE"
  android:value="true" />
```

Then, elsewhere in your code, when the tile content needs to be updated, you can call the `static requestListeningState()` method on `TileService`, to tell Android that you want it to poll your active `TileService` for an update.

And, Of Course, Bugs

Note that if you change your icon or caption after shipping your app, the user will not see the changes if they already have your tile in their notification shade. They would have to remove and re-add the tile to pick up the new icon and/or caption. This is due to this bug on Android 7.x and this bug on Android 8.x.

Only Enabling on Android 7.0+

On Android 6.0 and older devices, our `TileService` is useless. There is no value in having it be enabled, and there is always a chance that some security flaw might exploit it. So, it is best to disable this service on older devices. That is what the `android:enabled="@bool/is_n"` attribute is for in the `<service>` element. This is set to `false` in `res/values/bools.xml` and `true` in `res/values-v24/bools.xml`. Hence, on Android 6.0 and older devices, the service will be disabled.

The Service

`ToggleTileService` extends `TileService` and is responsible for dynamically changing the tile and responding to clicks on the tile:

```
package com.commonsware.android.sawmonitor;

import android.annotation.TargetApi;
import android.content.SharedPreferences;
import android.graphics.drawable.Icon;
import android.os.Build;
import android.preference.PreferenceManager;
import android.service.quicksettings.Tile;
import android.service.quicksettings.TileService;

@TargetApi(Build.VERSION_CODES.N)
public class ToggleTileService extends TileService {
  private SharedPreferences prefs;

  @Override
  public void onStartListening() {
    super.onStartListening();
```

```
    updateTile();
  }

  @Override
  public void onClick() {
    super.onClick();

    boolean isEnabled=getPrefs().getBoolean(MonitorApp.PREF_ENABLED, false);

    getPrefs()
      .edit()
      .putBoolean(MonitorApp.PREF_ENABLED, !isEnabled)
      .apply();
    updateTile();
  }

  private void updateTile() {
    Tile tile=getQsTile();

    if (tile!=null) {
      boolean isEnabled=getPrefs().getBoolean(MonitorApp.PREF_ENABLED, false);
      int state=isEnabled ? Tile.STATE_ACTIVE : Tile.STATE_INACTIVE;

      tile.setIcon(Icon.createWithResource(this,
        R.drawable.ic_new_releases_24dp));
      tile.setLabel(getString(R.string.app_name));
      tile.setState(state);
      tile.updateTile();
    }
  }

  private SharedPreferences getPrefs() {
    if (prefs==null) {
      prefs=PreferenceManager.getDefaultSharedPreferences(this);
    }

    return(prefs);
  }
}
```

(from Introspection/SAWMonitor/app/src/main/java/com/commonsware/android/sawmonitor/ToggleTileService.java)

As with many specialized `Service` subclasses, the API that you need to implement and consume for a `TileService` does not bear much resemblance to the regular `Service` API. You are welcome to override `onCreate()` and `onDestroy()` if needed, though `ToggleTileService` did not really need either of those.

If and when the user adds your tile to their notification shade, by default, you will be called with `onStartListening()`. There is also a corresponding `onStopListening()`. In between those two events, the user has the notification shade open, and so if you have changes that you need to publish to the tile, you should do so. For example, you might register some sort of event listener in `onStartListening()` (e.g., for WiFi signal strength changes) and unregister that listener in `onStopListening()`. While the listener is registered, if there is an event that needs to be reflected in the tile, your `TileService` might update that tile.

In this case, `onStartListening()` just updates the tile with the current state, in the private `updateTile()` method. `getQsTile()` returns a `Tile` object representing the current tile state, which you can examine and modify as needed. `getQsTile()` will return `null` if you cannot update the tile right now, for whatever reason.

In the case of `ToggleTileService`, we want the tile to reflect the state of the `enabled` `boolean` value in `SharedPreferences`. So, we lazy-load the `SharedPreferences` and see what `enabled` is. From there, we derive a `state` value, choosing between `STATE_ACTIVE` and `STATE_INACTIVE`. There are three possible states to choose from:

- `STATE_ACTIVE` is the normal state, indicating that the tile should be displayed normally and should support click events
- `STATE_INACTIVE` is the same, except that the tile should be displayed in an "inactive" style (e.g., dimmed)
- `STATE_UNAVAILABLE` — the default state until you indicate otherwise – indicates that the tile is disabled and will not respond to click events

`ToggleTileService` then updates the `Tile` with that state, along with setting the icon and label. Those happen to be set to the same values as are defined in the manifest, so this work is superfluous and here only for illustration. Note that `setIcon()` takes an `Icon` object, which can be created from a wide range of sources, including resources and local files.

Once you have the `Tile` configured to your liking, call `updateTile()` to push the changes over to the system, which will update the tile in the notification shade to match.

If the user clicks on the tile, you will be called with `onClick()`, where you can take whatever action makes sense. In this case, we just want to update the `SharedPreferences` to toggle the `enabled` value, then update the tile to match.

The User Experience

When the user installs an app that has a `TileService`, the tile is not automatically put in the user's notification shade. Instead, it *allows* the user to add the tile if the user wants to.

If the user opens the notification shade, an "Edit" button should appear:

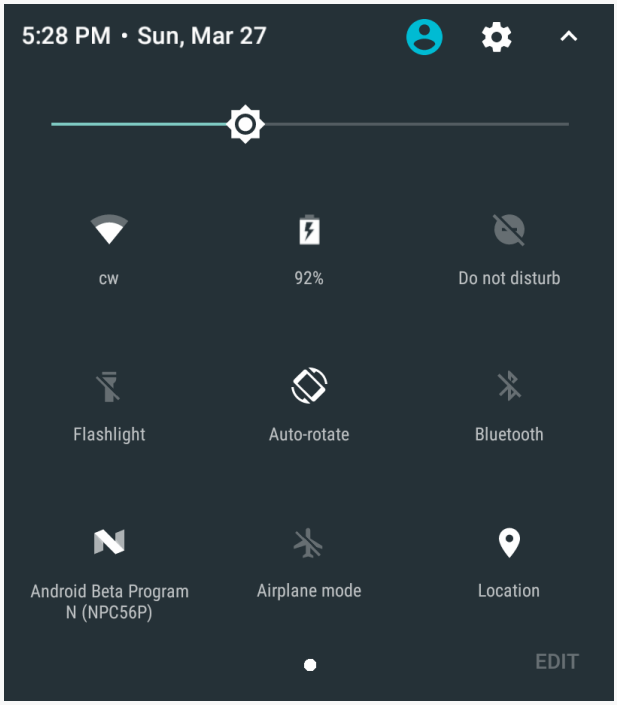


Figure 913: Notification Shade on Nexus 9 Running Android 7.0

Tapping "Edit" brings up the tile roster editor:

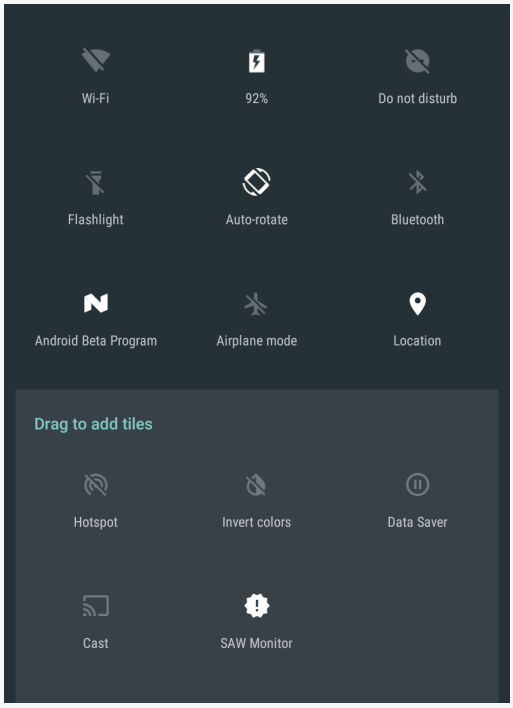


Figure 914: Android 7.0 Tile Roster Editor

The tile for the newly-installed app will appear. Since the `TileService` has never been invoked, the tile will display the icon and label from the `<service>` element.

The user can drag and drop from the “Drag to add tiles” area into the mock notification shade itself. Upon closing the editor, the new tile will appear in the notification shade

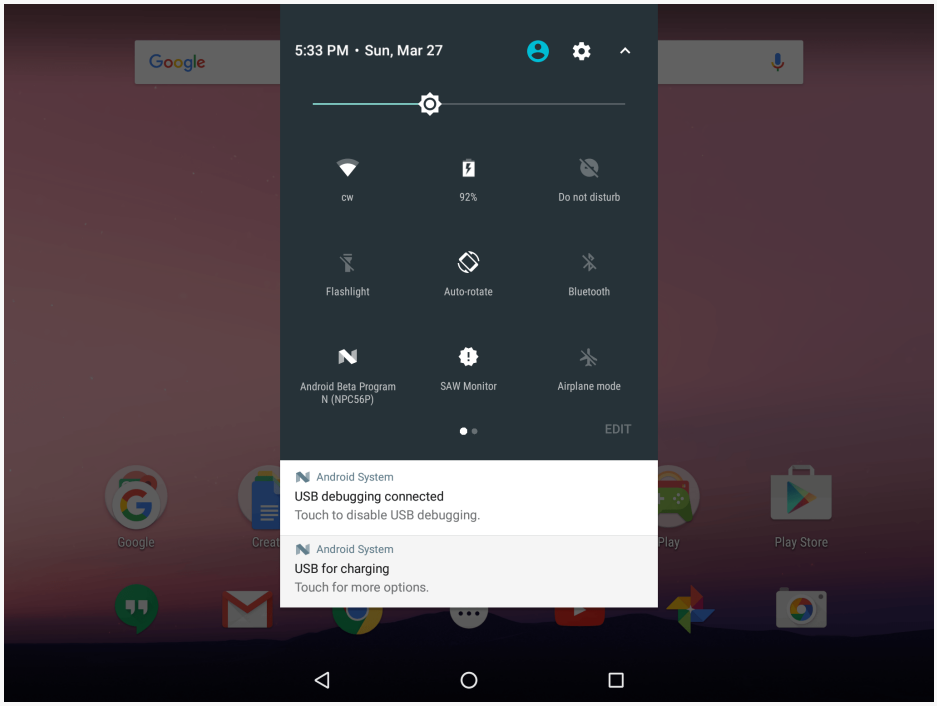


Figure 915: Notification Shade with SAW Monitor Tile

At this point, the tile should have been updated by the `TileService`. If the user later returns to the notification shade, the `TileService` will get another shot to update the tile via `onStartListening()`, and so forth.

If the user wants to, the user can return to the editor and drag the existing tile out of the mock notification shade back into the "Drag to add tiles" area, thereby removing it.

The Other Features and Limitations

Here are some other items of note related to tiles and `TileService`:

- You can determine if the device is locked by calling `isLocked()` on the `TileService`.
- You can respond to a tap on the tile by showing a dialog (`showDialog()`), launching an activity (`startActivityAndCollapse()`), or asking the user to unlock the device first (`unlockAndRun()`).
- If your tile may show sensitive data, `isSecure()` will tell you if your tile is visible in some "secure state" and therefore whether it is safe to show that sensitive data.

- Each `TileService` has only one tile. For most apps, this will be plenty. But, if for some reason your app needs multiple tiles, you will need multiple `TileService` implementations, one per tile.

Installing Packages

To install an app from an APK, you can use `ACTION_INSTALL_PACKAGE` on API Level 14+, and `ACTION_VIEW` on older devices. An `Intent` with one of those actions, and a `Uri` pointing to an APK, will lead the user to install the app... if the user grants our app the ability to install apps.

Prior to Android 8.0, the user would grant the ability to install "non-market apps" globally. This was simple but not especially secure, as any app could then ask to install other apps. While the user would be involved in the installation process — via confirmation dialogs and such — this still was not ideal.

On Android 8.0+, the user flow is a bit different, as now rather than enabling app-installation rights to all apps via Settings, the user grants it on a per-installer basis.

We can determine whether or not we have app-installation rights by calling `canRequestPackageInstalls()` on a `PackageManager`. However:

- This method is only available on Android 8.0+
- Apps need to hold the `REQUEST_INSTALL_PACKAGES` permission to make this call
- This value does you little good, other than to perhaps warn the user as to what is coming, as there is no way to request app-installation rights other than to try installing an app

The `Introspection/AppWrangler` sample project has an overflow menu with an "Install..." item in it. Tapping that will bring up an `ACTION_OPEN_DOCUMENT` screen, for you to find an APK on your device or emulator. When we get the `Uri` in `onActivityResult()`, we:

- Call `canRequestPackageInstalls()` and show a `Toast` if we do not already have app-installation rights
- Start the `ACTION_INSTALL_PACKAGE` activity regardless of the `canRequestPackageInstalls()` state

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
```

```
                                Intent data) {
  if (requestCode==REQUEST_OPEN && resultCode==RESULT_OK) {
    if (getPackageManager().canRequestPackageInstalls()) {
      Toast.makeText(this, R.string.msg_install_perm, Toast.LENGTH_LONG).show();
    }

    Intent i=new Intent(Intent.ACTION_INSTALL_PACKAGE)
      .setData(data.getData())
      .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

    startActivity(i);
  }
}
```

(from Introspection/AppWrangler/app/src/main/java/com/commonsware/android/appwrangler/MainActivity.java)

What the user sees — other than the `Toast` — is a dialog confirming whether or not installing an app is what the user has in mind:

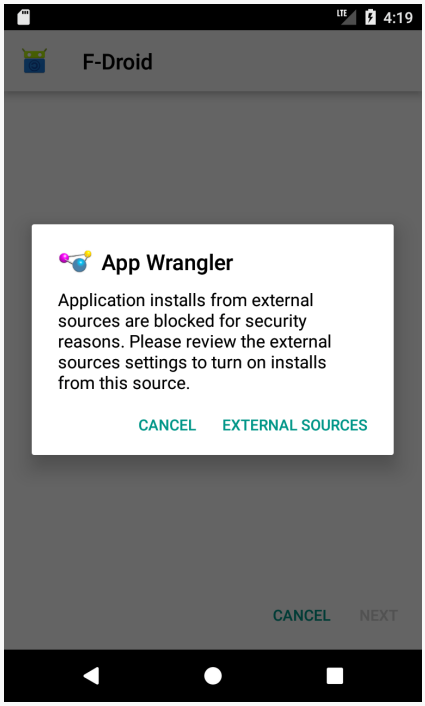


Figure 916: Android 8.0 Emulator, Initial App Installation Dialog

If the user taps “External Sources”, a screen dedicated to your app appears:

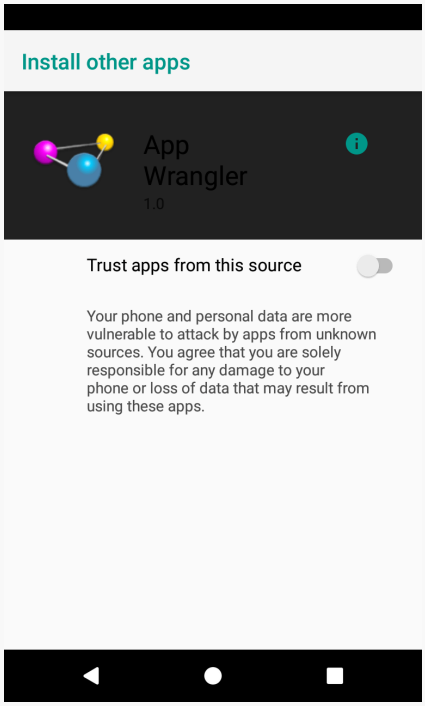


Figure 917: Android 8.0 Emulator, Trust Screen in Settings

(the dark gray background on the top banner appears on the emulator, but not on hardware)

If the user checks the “Trust apps from this source” switch and presses BACK, ideally the user is then taken to the normal app installation flow:

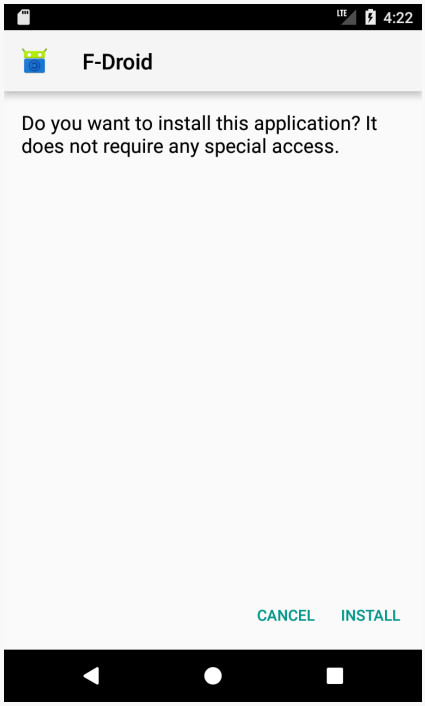


Figure 918: Android 8.0 Emulator, Installing F-Droid

Deleting Packages

Similarly, to install an app from an APK, you can use `ACTION_UNINSTALL_PACKAGE` on API Level 14+, and `ACTION_DELETE` on older devices. An `Intent` with one of those actions, and a `Uri` pointing to an APK, will ask the user if they want to uninstall the app.

None of that has changed.

There is an `REQUEST_DELETE_PACKAGES` permission, and its documentation indicates that you need this to use `ACTION_UNINSTALL_PACKAGE`. This is not the case.

Detecting Changes in Packages

Sometimes, your app might need to know if apps are installed or removed.

On Android 7.1 and older, if we want to find out about changes in the mix of installed packages, we would listen for `ACTION_PACKAGE_ADDED` and `ACTION_PACKAGE_REMOVED`.

However, those are implicit broadcasts, and we cannot register for those in the manifest on Android 8.0+. As a result, we can only listen to those broadcasts when we already have a process running for other reasons and can use `registerReceiver()`. That is fine as far as it goes, but it means that we may not find out about every package change.

The workaround for this is to call `getChangedPackages()` on `PackageManager` on Android 8.0+. This returns a `ChangedPackages` object, listing the packages that have changed.

The scope of the change is based on a sequence number. This starts at zero, when the device is booted. Every change to the mix of installed packages should trigger a new sequence number. You pass in the sequence number of the last `getChangedPackages()` call that you made, or `0` if you have not called it before. The `ChangedPackages` will give you the list of packages changed between your passed-in sequence number and the current state. The `ChangedPackages` also has the current sequence number. The idea is that you can persist that sequence number, so when your app runs, you can find out about package changes that went on while your process was not around.

The `AppWrangler` sample app demonstrates this, though not using persistence, as it is not needed here.

The `AppWrangler` UI is dominated by a list of installed apps. That list is populated at the outset by a call to `refresh()` from `onResume()`:

```
private void refresh(boolean toast) {
  if (lastPackageSequenceNumber==-1) {
    ChangedPackages delta=pm.getChangedPackages(0);

    lastPackageSequenceNumber=(delta==null) ? 0 : delta.getSequenceNumber();
    populateList();
  }
  else {
    ChangedPackages delta=pm.getChangedPackages(lastPackageSequenceNumber);

    if (delta!=null && delta.getSequenceNumber()>lastPackageSequenceNumber) {
      populateList();

      if (toast) {
        Toast.makeText(this, R.string.msg_refresh_ack,
          Toast.LENGTH_SHORT).show();
      }
    }
```

```
      else if (toast) {
        Toast.makeText(this, R.string.msg_refresh_nack,
          Toast.LENGTH_LONG).show();
      }
    }
  }
```

(from Introspection/AppWrangler/app/src/main/java/com/commonsware/android/appwrangler/MainActivity.java)

`lastPackageSequenceNumber` is initialized to `-1`, meaning that we have never tried getting our packages yet. In that case, we call `getChangedPackages()` to get the now-current sequence number, passing in `0` as the starting sequence number. Since `getChangedPackages()` might return `null`, we either read the sequence number out of the `ChangedPackages` or use `0` for the new `lastPackageSequenceNumber` value. Then, we call `populateList()`, which uses `PackageManager` and `getInstalledApplications()` to populate the `ListView`.

There is a "refresh" action bar item, which also triggers a call to `refresh()`. In this case, `lastPackageSequenceNumber` should be something other than `-1`, courtesy of our original `refresh()` call triggered by `onResume()`. If `lastPackageSequenceNumber` is not `-1`, we call `getChangedPackages()`, supplying `lastPackageSequenceNumber` as the starting point. If we get a `ChangedPackages` back, and the new sequence number is higher, we know that there was a package change that went on "behind our backs", so we update the list. If we were using `RecyclerView`, we might use the list of packages in the `ChangedPackages` to make the specific modifications to our list that are required — since that is not really an option with `ListView`, we just reload the whole list.

If, with the `AppWrangler` app in the foreground, you install an app from the command line using `adb install`, then click the "refresh" action bar item, you should see a `Toast` indicating that there were changes in the packages, and the list should update to show the newly-installed app.

Replacing App Code Dynamically

You write your app. You test your app. You upload your app to your favorite distribution channels. Your users download and use your app. You get bug reports and feature requests, to go along with your existing plans for the app.

So, you update your app. You test your app. You upload your app to your favorite distribution channels. Your users download and use your updated app. You get more bug reports and feature requests, to go along with your existing plans for the app.

So, once again, you update your app. You test your app. You upload your app to your favorite distribution channels. Your users download and use your updated app. You get still more bug reports and feature requests, to go along with your existing plans for the app.

Or, as the famous shampoo instructions go: lather, rinse, repeat.

Some developers have been working on breaking the cycle, specifically by delivering updates directly, bypassing the Play Store or other distribution channels. This offers a lot of potential for speed of updates and flexibility. It comes with some fairly substantial costs in terms of complexity and security. This chapter will explore some options in this area, so you can understand what is possible and what those costs and benefits are.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

The major example app in this book is based on the OkHttp3 sample app from the chapter on Internet access. The prose and code in this chapter will make a *lot* more sense if you are familiar with the Internet chapter's examples.

The major example also uses `TrustManagerBuilder`, discussed in the chapter on SSL support.

Typical Objectives

There are any number of possible reasons why a developer may want to find ways to update the app's code without having to ship a fresh APK.

Continuous Deployment

Many mobile developers have a history with Web development. Many firms have adopted a "continuous deployment" approach with their Web sites or Web apps. Here, rather than there being sporadic sweeping upgrades to the Web sites and apps, small changes are applied frequently, sometimes multiple times per day. Partially, the idea is to get good at deployment through repetition and practice, to reduce the odds of any given deployment running into a problem. Partially, the idea is to keep refreshing the Web site or Web app, so regular users get constant small improvements, as opposed to having to deal with big wrenching changes in UI or UX when major updates are deployed.

It stands to reason that firms with a history of continuous Web deployment would be interested in continuous mobile deployment as well. However, in Android, distributing a new APK is not seamless. While the work on the firm's behalf may be small, every user has to apply the upgrade. Continuous deployment, from the user's standpoint, means continuous aggravation, as the Play Store or other distribution channel pesters them about a new app update. Continuous deployment in Android is only practical when many of the upgrades can be done without replacing the APK.

Forced Immediate Updates

Many times, a mobile app is a peer of a Web app, both using the same server, just via different interfaces. The mobile app might be using REST or GraphQL endpoints to retrieve pure data, while the Web app is serving standard HTML/CSS/JavaScript.

One challenge in this case is keeping everything consistent with respect to business rules.

For example, perhaps the original business rule for passwords was "no longer than 8 characters", because some manager was worried that the server might run out of disk space due to people having longer passwords. Eventually, security engineers pounded into the manager's head that storing salted hashes of passwords in the database meant that the length of the password itself was immaterial in terms of server space. So, the business rule was amended to raise the password limit to 32 characters. On the Web app, this change might take seconds to implement and deploy. In the Android app, while the change might take seconds to implement, deployment poses two challenges:

1. You have to ship a fresh APK with the changed code, and that takes time
2. Not everybody updates the app right away, or ever, choosing to ignore those pesky notifications from the Play Store or other distribution channel

All the older editions of the app being used have the older business rule. So, a user goes into the Web site and changes their password to be 14 characters long... and now cannot log into the Android app, because their older Android app limits passwords to 8 characters.

Perhaps continuous deployment of new features and functions is beyond what you are interested in. However, this sort of business-rule change is the kind of thing that you might want to try to keep in sync between different editions of your app (Android, iOS, Web, etc.), and the coarse-grained nature of APK updates makes that difficult. Being able to deploy fresh business rules on the fly, without going through an APK update, would help.

Avoid Security Review

Not every reason for dynamically updating the code is a positive one.

The Play Store — and hopefully most other professional app distribution channels — performs some automated scans for malware. They do so on the APKs that are uploaded into the channels' developer portals. Some, like the Play Store, also offer scanning services for APKs installed via other means (a.k.a., "sideloaded" apps), but once again, the scan is of the APK being installed.

A sophisticated malware distributor can apply the same basic techniques for deploying on-the-fly feature or business rule changes for deploying on-the-fly malware. The objective here is to bypass being detected by the APK security scans, simply by not having the malware be in the original APK, but rather via some dynamically-applied code that is loaded later.

The Challenges

Implementing some sort of dynamic code replacement is far from simple. Even at the level of a book example, there are "a lot of moving parts", let alone for a production-grade app.

Security

The area that tends to raise the most issues is security. We want users to get new code, but only the *right* new code, not new code that actually is malware in disguise.

MITM Attacks

Often times, the plan for distributing the dynamic code updates is to do so over the Internet. For example, the app might download the update from a Web server maintained by the app's developer.

Any time that you transfer data over the Internet, there is a risk of a "MITM" attack (variously defined as "man in the middle" or "Martian in the middle", depending on your attitudes regarding Martians).

For example, perhaps the device is connected to the Internet via some public WiFi hotspot when the app goes to download an update. However, that hotspot was hacked, with a proxy server spying on all Web traffic. That proxy might replace the real dynamic code update with an altered one containing malware. Unless the app developer takes specific steps in the app to detect this situation and defend against it, the app will blindly use the malware-laden update, probably to the user's regret.

Code Replacement "At Rest"

A MITM attack replaces code "in motion", as it is being distributed over the Internet. It is also possible for an attacker to replace your code "at rest", as usually these dynamic code updates get saved in files for future use.

If those files are on external storage, any app can replace them with impunity, so long as the user has granted `WRITE_EXTERNAL_STORAGE` access to the attacker.

If those files are on removable storage, any app can replace them with impunity, so long as the user has convinced the user to grant the app write access to those files.

That could be on an individual basis via `ACTION_OPEN_DOCUMENT` or in bulk via `ACTION_OPEN_DOCUMENT_TREE`.

The conventional wisdom is that these sorts of dynamic code updates should be stored on internal storage, which ordinary other apps cannot access. However:

- On a rooted device, an attacker might still be able to replace this code (though the attacker could replace your entire APK in that situation)
- While outside apps cannot directly access your internal storage, they might be able to exploit bugs in your app to get *you* to attack your own files

The quintessential example of this comes from the backport of multidex.

Multidex is a capability, native to Android 5.0+, where the Java code for an app can be split into multiple DEX files, rather than the single DEX file found in Android apps since Android 1.0, where a DEX file contains the Dalvik bytecode for the app. This is to bypass certain limits on application complexity (the "64K DEX method reference limit").

When multidex was introduced, a backport was provided. Developers could add a support library to their app and add in a bit of boilerplate code to the app. Then, the build tools can create multidex APKs that can run on older environments. What the support backport does is unpack the additional DEX files from APK, then use the same techniques that we can for dynamic code updates to arrange to load classes from those additional DEX files.

On the surface, the behavior is the same between the native multidex implementation and the backport: the app is distributed with multiple DEX files. However, there is one key difference: the DEX files in the backport can be read from and written to by the app itself, whereas with native multidex, the app cannot modify those DEX files. This is a side effect of the app having to be the one to extract and save those DEX files from the APK file — if the app can do that, by definition the app can go in and change those DEX files later (e.g., on an upgrade).

This, by itself, is not a problem. However, it does open the opportunity for attackers to exploit *another* bug in the app, using it to trick the app into replacing those DEX files with malware-laden alternatives. For example, apps that improperly expand ZIP archives could be tricked into expanding a ZIP archive that overwrites the DEX files with alternatives.

Delivery

Getting dynamic code updates to the user itself can be a challenge.

If you are using the Internet for dynamic code distribution, on the plus side, there are lots of tools and techniques at your disposal. However, too many developers think that the Internet is reliable everywhere and behaves consistently, and that is not the case. DNS resolution issues, WiFi captive portals, "great firewalls" erected by various countries, and the like make getting material to users over the Internet more difficult than you might think. Obviously, it can be done. However, do not assume that it will be consistent or reliable.

If you are planning on using some other means to get the dynamic code updates to users... now you have to invent your own infrastructure. For apps with few users, and ones that you can communicate with regularly, this may not be a problem. The broader your user base, and the less reliable your communications are with those users, the more likely it is that getting dynamic code updates to those users will prove to be troublesome.

The Scripting Solution

There are many scripting languages available for Android. Some run on the virtual machine, while others are implemented using native libraries and the NDK. In either case, you get a Java bridge to be able to invoke scripts on demand. Frequently, you get a reverse bridge, where the scripts can call Java code from your app.

You could implement the portions of your app that you want to replace dynamically in a scripting language. Then, updating the app is "merely" a matter of downloading and using a new script.

Example: Square and Duktape

At the 2016 droidcon NYC conference, Matt Precious spoke about how Square is taking this approach.

For business rules (e.g., how long can a password be?), rather than hard-code the values in Java, or rather than hard-code the possible rules with values being downloaded (e.g., via JSON), Square implements the rules and values in JavaScript.

For their Web app, they can just use the JavaScript directly. For Android, they use duktape, an embeddable JavaScript interpreter implemented in C.

Specifically, the Square team wrote `duktape-android`, an Android-friendly packaging of duktape, to make it easy to execute scripts, get responses, allow scripts to call back into the app's Java code, and so forth.

By implementing the business rules in one place, they have guaranteed synchronization between the two app platforms. Their duktape-enhanced apps can download the latest rules on startup and apply them as needed. Appropriately "minified" and compressed JavaScript is fairly tiny, adding little Internet burden to the app or the app's users.

On the other hand, `duktape-android` itself is a ~2MB AAR (in version 1.0.0), and so there is some amount of cost to the user at the point when a duktape-enhanced app is installed.

Security, as usual, is a challenge here. Securing this JavaScript "in motion" (i.e., during the download process) can be handled through advanced SSL techniques, such as certificate pinning or limiting valid certificate authorities. However, a plain JavaScript file cannot readily be digitally signed to help provide evidence of authorship or show signs of tampering. Ideally, these scripts would be distributed in some container that *can* be signed, such as a ZIP file.

The Hybrid Solution

Hybrid app frameworks allow developers to create Android apps — and apps for other platforms — using Web technologies like HTML, CSS, and JavaScript. Perhaps the best known of these is Adobe's PhoneGap, based on the open source Apache Cordova project. However, there are a variety of alternatives.

These frameworks have a common approach. They provide a shell of an Android app, enough to set up a `WebView` and, from there, render the UI defined in the HTML/CSS/JS. The `WebView` is hosted in an activity that is provided by the framework, one that extends the `WebView` to allow Android events (e.g., lifecycle methods) to be passed into the JavaScript of the page and for JavaScript to call back into Android (e.g., to access contacts).

Usually, these frameworks package the HTML/CSS/JS in the Android APK as assets, as `WebView` can easily load content from assets, and it delivers the Web content along

with the rest of the app. However, little is stopping one of these frameworks from allowing developers to publish fresh HTML/CSS/JS somewhere. The framework code could monitor a configurable URL for updates, download those updates, and switch over to them, all without requiring a fresh APK to be distributed.

The same sorts of issues are seen here as with the scripting solution:

- You would want to use certificate pinning or other advanced SSL techniques to ensure that you are downloading the updates from the proper location
- You would want to distribute those updates in a digitally-signed container, so the app could validate that the updates came from the app developer and were not modified along the way

The Patch Solution

A variety of libraries have been released that offer a "hot-fix" capability for Android apps. Led by Tencent's Tinker, these libraries advertise the ability to update an APK's code and resources without having to ship a full APK update.

For example, with Tinker, tools provided via a Gradle plugin allow you to create patches based upon the difference between a new and old APK. By adding their library and tweaking your app a bit to use the library, you can arrange for the library to get the patch and apply it to your app.

Tinker, like Alibaba's AndFix, byteam's delta, eleme's Amigo, and others suffer from a general lack of documentation. It is unclear to what extent they validate that the patch has not been tampered with (in motion or at rest).

How Does This Relate to Instant Run?

Google announced Instant Run to great fanfare in 2015, as a new feature for Android Studio 2.0. Instant Run works similarly to these patch libraries: when you "run" the app from the IDE, and the app is already installed on the device, Instant Run creates and sends over a patch rather than a full APK. The idea is to cut the data transfer time, avoiding re-sending bytes that are not needed, and avoiding having Android have to go through an app installation process.

Many developers have encountered problems with Instant Run and have disabled it as a result. The fact that Google cannot readily perform this patching – even in a

constrained development environment — demonstrates the difficulty in getting this sort of approach to work widely in production.

A DIY Solution

Maybe you do not want to write your code in JavaScript and try gluing it into the rest of your app, the way Square did with `duktape-android`. And, perhaps your use case does not quite fit the existing patch solutions outlined earlier in this chapter. So, you want use the techniques from those patch solutions, but in your own way.

This is very possible, but it is very messy.

What We Want

Let's think of this in terms of app extensions, adding new functionality to an existing Android app without having to deliver that new functionality in the form of an update to our app's APK.

In particular, let's allow an app developer to:

- Create one or more extensions to their app (referred to here as the "host" or "host app")
- Use Java interfaces to define the relationship between the host and the extension, to simplify writing the host and writing the extension
- Deliver the extension to users by whatever means the developer wants (though the sample app will demonstrate downloading extensions via HTTPS)
- Load these extensions on the fly, without users necessarily having to be involved in the process
- Store these extensions wherever the app has filesystem access (though, as previously noted, for security reasons, storing the extensions on internal storage is almost universally the right answer)
- Validate that the extensions have not been tampered with or replaced by malware
- Develop extensions through a reasonably graceful process, using Android Studio and Gradle, without having to resort to lots of custom tricks
- Deliver other sorts of files along with the extensions, for possible use by the host app
- Reset the environment, removing extensions and related materials as desired

Introducing the Sample App

Back in the chapter on Internet access, we reviewed several versions of a sample app that loaded the latest questions in the `android` tag from Stack Overflow. The `DynCode/ListLoader` sample project is based on those, specifically the one that used OkHttp to retrieve the questions in JSON format and Gson to parse those into objects.

The main app will be responsible for displaying a list of things. What those things are will come from an extension. Hence, while the extension that we develop here will load `android` questions from Stack Overflow, a different question could load things from any desired data source, local or remote, so long as the extension implements the designated API.

Defining the Extension API

It helps, when implementing this sort of extension system, to have a common API that both the host app and the extensions agree upon. And, for reasons that will become apparent as we move along, it will be easiest to have that API implemented in an Android library module, to be shared among other modules.

The sample project's `api/` module defines this extension API. It does so in the form of two interfaces and one class.

The job of an extension is to load things. So, we have a `Thing` interface, describing what information the host app needs from the extension for a thing:

```
package com.commonsware.android.dyncode.api;

public interface Thing {
  String getTitle();
  String getLink();
}
```

(from DynCode/ListLoader/api/src/main/java/com/commonsware/android/dyncode/api/Thing.java)

All we need is for a `Thing` to give us some sort of title (to display in list rows) and a URL (to convert into a `Uri` and use with an `ACTION_VIEW` `Intent`, to view more details about this thing).

The extension is really responsible for delivering us a `List` of `Thing` objects. That is handled via some extension-supplied implementation of a `ThingsLoader`:

```
package com.commonsware.android.dyncode.api;

import java.util.List;

public interface ThingsLoader {
  void startAsyncLoad();
  List<Thing> getThings();
}
```

(from DynCode/ListLoader/api/src/main/java/com/commonsware/android/dyncode/api/ThingsLoader.java)

The host app should call `startAsyncLoad()`, which will load the things in a background thread supplied by the extension. The extension and the host app will share greenrobot's EventBus for return communications, and the host app will be told when the things are ready via a `ThingsLoadedEvent` delivered via the event bus:

```
package com.commonsware.android.dyncode.api;

import java.util.List;

public interface ThingsLoader {
  void startAsyncLoad();
  List<Thing> getThings();
}
```

(from DynCode/ListLoader/api/src/main/java/com/commonsware/android/dyncode/api/ThingsLoader.java)

Any time after that, the host app can call `getThings()` on the `ThingsLoader` to get the list of things to show in the list.

The sample project also has a traditional Android Studio `app/` module, which depends upon the `api/` module, along with the EventBus artifact and a couple of CWAC libraries:

```
repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'com.commonsware.cwac:security:0.8.0'
    implementation 'com.commonsware.cwac:netsecurity:0.2.0'
    compile project(':api')
    implementation 'de.greenrobot:eventbus:2.4.0'
}
```

(from DynCode/ListLoader/app/build.gradle)

We will look at how the `app/` module uses the extension API later in this chapter.

Writing the Extension Code

Of course, somewhere, we have to load some things, in the form of having a `ThingsLoader` load some `Things`.

The `impl/` module in the sample project contains code, derived from the original Stack Overflow app examples, that implements `Thing` and `ThingsLoader`.

Specifically, the `Question` class is now an implementation of `Thing`:

```
package com.commonsware.android.dyncode.impl;

import com.commonsware.android.dyncode.api.Thing;

public class Question implements Thing {
  String title;
  String link;

  @Override
  public String toString() {
    return(getTitle());
  }

  @Override
  public String getTitle() {
    return(title);
  }

  @Override
  public String getLink() {
    return(link);
  }
}
```

(from DynCode/ListLoader/impl/src/main/java/com/commonsware/android/dyncode/impl/Question.java)

Not surprisingly, `QuestionsLoader` is our `ThingsLoader` implementation, using OkHttp and Gson:

```
package com.commonsware.android.dyncode.impl;

import android.util.Log;
```

```
import com.commonsware.android.dyncode.api.Thing;
import com.commonsware.android.dyncode.api.ThingsLoadedEvent;
import com.commonsware.android.dyncode.api.ThingsLoader;
import com.google.gson.Gson;
import java.io.BufferedReader;
import java.io.Reader;
import java.util.ArrayList;
import java.util.List;
import de.greenrobot.event.EventBus;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class QuestionsLoader implements ThingsLoader {
  private SOQuestions rawResult;
  private List<Thing> things;

  @Override
  public void startAsyncLoad() {
    new LoadThread().start();
  }

  @Override
  public List<Thing> getThings() {
    if (things==null) {
      things=new ArrayList<>();

      for (Question item : rawResult.items) {
        things.add(item);
      }
    }

    return(things);
  }

  private class LoadThread extends Thread {
    static final String SO_URL=
        "https://api.stackexchange.com/2.1/questions?"
            + "order=desc&sort=creation&site=stackoverflow&tagged=android";

    @Override
    public void run() {
      try {
        OkHttpClient client=new OkHttpClient();
        Request request=new Request.Builder().url(SO_URL).build();
        Response response=client.newCall(request).execute();

        if (response.isSuccessful()) {
```

```
          Reader in=response.body().charStream();
          BufferedReader reader=new BufferedReader(in);

          rawResult=new Gson().fromJson(reader, SOQuestions.class);
          reader.close();

          EventBus
            .getDefault()
            .post(new ThingsLoadedEvent(QuestionsLoader.this));
        }
        else {
          Log.e(getClass().getSimpleName(), response.toString());
        }
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
    }
  }
}
```

(from DynCode/ListLoader/impl/src/main/java/com/commonsware/android/dyncode/impl/QuestionsLoader.java)

`startAsyncLoad()` just kicks off a `LoadThread` to load the questions. Gson, however, is going to give us an `SOQuestions` structure back, to mirror the JSON format used by the Stack Exchange API. `SOQuestions` holds a `List` of `Question` objects:

```
package com.commonsware.android.dyncode.impl;

import java.util.List;

public class SOQuestions {
  List<Question> items;
}
```

(from DynCode/ListLoader/impl/src/main/java/com/commonsware/android/dyncode/impl/SOQuestions.java)

Unfortunately, due to a quirk in Java's type checking, we cannot return a `List` of `Question` objects from `getThings()`, even though `Question` implements `Thing`. Instead, we need to convert the `List` of `Question` objects into a `List` of `Thing` objects, which we do (and cache, to save going through that work again if we are called with `getThings()` again).

The `impl/` module is a library module. The idea is that you could attach the `impl/` module directly to the host app (`app/` module) during development, so you can test this code the same as you would pretty much anything else.

Dealing with Dependencies

One key issue with creating the implementation comes with its use of libraries. *If* your extension will use the some of the same libraries as does your app, *and* you are supporting Android 4.4 and older devices, then you need to take special care when creating your `dependencies` closure. Specifically, for those libraries that are in common between the app and the extension, use `compileOnly` instead of `implementation` for the `dependencies`.

For example, our `impl/` uses two libraries that the main app also uses:

- greenrobot's EventBus
- our `api/` library

The app will provide the code for these libraries for both the app itself and for the extension. We do not need to package the code for these libraries in the extension, and even if we did, our extension would be unusable on Android 4.4 and older due to limits in the Dalvik classloaders. So, our `impl/build.gradle` file specifies those dependencies as `compileOnly`:

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion 24
    buildToolsVersion "25.0.3"

    defaultConfig {
        minSdkVersion 15
        targetSdkVersion 24
    }
}

dependencies {
    provided 'de.greenrobot:eventbus:2.4.0'
    compile 'com.google.code.gson:gson:2.4'
    compile 'com.squareup.okhttp3:okhttp:3.8.0'
    provided project(':api')
}
```

(from DynCode/ListLoader/impl/build.gradle)

`compileOnly` means that the Java build tools should compile against the classes in the libraries, but that the libraries themselves should not be included in the APK. By

contrast, `implementation` says that the classes are available at compile time *and* are packaged into the APK.

The Gson and OkHttp libraries use `implementation`, as the app does not have those libraries, but the extension uses them.

Packaging an Extension

Now, we need to create some sort of file that contains the DEX compiled code from our `impl/` library module. The AAR that is the natural compiled output of a library module will not work, as it contains an ordinary Java JAR with Java bytecodes, not DEX bytecodes. Android itself only knows how to work with DEX bytecodes, relying on development tools to cross-compile Java bytecodes into DEX equivalents.

The simplest solution for packaging the extension, as it turns out, is to just create an APK for the extension itself.

So, the fourth (and final) module in the sample project is `ext/`. This is a regular Android application module, pulling in the `impl/` library module as a dependency:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 24
    buildToolsVersion "25.0.3"

    defaultConfig {
        applicationId "com.commonsware.android.dyncode.ext"
        minSdkVersion 15
        targetSdkVersion 24
    }
}

dependencies {
    compile project(':impl')
}
```

(from DynCode/ListLoader/ext/build.gradle)

Other than a stub manifest, the `ext/` module has nothing else of its own. You can build this module normally, for either `debug` builds or for `release` builds (if you supply signing information). The APK will be the file that you will want to get to users of the host app by one means or another — the sample app will use a Web server, as you will see shortly.

Enter the Thunk

At this point, we have an extension, packaged in an APK, that implements the `Thing` and `ThingsLoader` APIs.

If we were to link the `impl/` library directly into the host app, we would be able to simply reference the `Question` and `QuestionLoader` classes, perhaps hiding them behind their `Thing` and `ThingsLoader` interfaces. That way, the host app could generically use `Thing` and `ThingsLoader` in most places, with only a bit of code knowing to wire in the `Question` and `QuestionLoader`.

However, we do not want to distribute the implementation in the host app; we want to distribute the implementation via the extension. So, our host app needs another way to work with a `ThingsLoader`.

Specifically, what we could really use is some way for the host app to still use the `ThingsLoader` interface, but where the implementation is coming from the extension, not from the host app's own code.

For that, the host app has `ThingsLoaderThunk`. This class implements the `ThingsLoader` interface. The thunk is responsible for:

- Getting the extension onto the device (e.g., via an HTTPS download)
- Validating the extension (to confirm it is not malware or tampered with)
- Loading the extension and getting access to the DEX-compiled classes inside of it
- Passing along the `ThingsLoader` calls to the `ThingsLoader` supplied via the extension

To do all that, the thunk will need a couple of pieces of information, such as:

- Where to download the extension from, and
- What Java class in the extension are we trying to load and use as a `ThingsLoader`

Where the host app gets that sort of information is up to the host app's developer. This might be metadata retrieved by a Web service, for example. Here, we will use somewhat more hard-coded approaches, to try to simplify this example a bit.

Delivering an Extension

The URL will come from an `EXTENSION_URL` global variable, accessed in our `app/build.gradle` file, poured into a `BuildConfig` field:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 24
    buildToolsVersion '26.0.2'

    defaultConfig {
        minSdkVersion 17
        targetSdkVersion 24
        buildConfigField "String", "EXTENSION_URL", '"'+EXTENSION_URL+'"'
    }
}

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}

dependencies {
    implementation 'com.commonsware.cwac:security:0.8.0'
    implementation 'com.commonsware.cwac:netsecurity:0.2.0'
    compile project(':api')
    implementation 'de.greenrobot:eventbus:2.4.0'
}
```

(from DynCode/ListLoader/app/build.gradle)

This `EXTENSION_URL` could come from a number of places, but the typical approach would be to define it in the project's `gradle.properties` file. We will see what that would look like later in this chapter.

Our `ThingsFragment` in the `app/` module is a `ListFragment` for showing the list of `Thing` objects that we get from the extension. It needs to work with the `ThingsLoader`, and since that `ThingsLoader` is in the extension, the `ThingsFragment` will instead set up a `ThingsLoaderThunk`, as part of its `onCreate()` processing:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    setRetainInstance(true);
    setHasOptionsMenu(true);

    try {
      loader=
        new ThingsLoaderThunk(getActivity(),
          BuildConfig.EXTENSION_URL,
          "com.commonsware.android.dyncode.impl.QuestionsLoader",
          this);
      loader.startAsyncLoad();
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception loading extension",
        e);
      Toast
        .makeText(getActivity(), "Exception loading extension",
          Toast.LENGTH_LONG)
        .show();
    }
  }
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsFragment.java)

`ThingsLoaderThunk` takes four constructor parameters:

- a `Context`
- the URL where the extension should be downloaded from
- the fully-qualified Java class name of the `ThingsLoader` in the extension
- a `ThingsLoaderThunk.Callback` implementation, which we will explore in detail shortly

`ThingsLoaderThunk`, as part of its work in the constructor, holds onto the URL in a field and creates a `TrustManagerBuilder`:

```
    this.url=new URL(BuildConfig.EXTENSION_URL);

    tmb=new TrustManagerBuilder().withConfig(ctxt,
      R.xml.extension_server);
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

`TrustManagerBuilder` is from the CWAC-NetSecurity backport of the Android 7.0 network security configuration subsystem.

Eventually, `ThingsLoaderThunk` will call a `downloadExtension()` method, which uses `HttpURLConnection`, along with the configured `TrustManagerBuilder`, to download the extension:

```
private void downloadExtension() throws Exception {
  HttpURLConnection c=
    (HttpURLConnection)url.openConnection();

  tmb.applyTo(c);

  FileOutputStream fos=
    new FileOutputStream(apkPath.getPath());
  BufferedOutputStream out=new BufferedOutputStream(fos);

  try {
    InputStream in=c.getInputStream();
    byte[] buffer=new byte[8192];
    int len;

    while ((len=in.read(buffer))>=0) {
      out.write(buffer, 0, len);
    }

    out.flush();
  }
  finally {
    try {
      fos.getFD().sync();
      out.close();
    }
    finally {
      c.disconnect();
    }
  }

  validateApk(apkPath);
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

The `R.xml.extension_server` resource used with the `TrustManagerBuilder` needs to contain whatever rules are appropriate for ensuring that we are downloading this file from the proper server. In particular, those rules should block MITM attacks using certificate pinning, limiting certificate authorities, or similar approaches. We will see this more later in this chapter.

We will also see the call to `downloadExtension()` in a later section.

Validating an Extension

One of the things that `downloadExtension()` does is call a `validateApk()` method. The idea is that this method will validate whether the APK has a valid digital signature and whether that signature matches the signature of the app itself. That way, we can be fairly certain that the extension is genuine and has not been modified by outside parties.

Fortunately, this is covered by `getPackageArchiveInfo()`, available on `PackageManager`. You supply `getPackageArchiveInfo()` with the path to the APK file, along with flags (e.g., `GET_SIGNATURES`). If the APK does not have a valid signature, `getPackageArchiveInfo()` returns `null`, and you know that you should not load the APK. If the APK is valid, you get a `PackageInfo` object back with the requested details filled in.

So, `validateApk()` calls a `getApkSignatures()` method to retrieve the signatures from the extension APK:

```
private Signature[] getApkSignatures(File apkPath) {
  PackageInfo info=
    pm.getPackageArchiveInfo(apkPath.getAbsolutePath(),
      PackageManager.GET_SIGNATURES);

  if (info==null) {
    throw new IllegalStateException("Extension APK could not be parsed");
  }

  return(info.signatures);
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

We retrieve a similar collection of `Signature` objects from our own app, up in the `ThingLoaderThunk` constructor:

```
  pm=ctxt.getPackageManager();
  ownSigs=
    Arrays.asList(pm
      .getPackageInfo(ctxt.getPackageName(), PackageManager.GET_SIGNATURES)
      .signatures);
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

`validateApk()` then just confirms that both collections have the same length and that each signature in the APK's collection matches one from the app itself:

```
private void validateApk(File apkPath) {
  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
    try {
      Signature[] fileSigs=getApkSignatures(apkPath);

      if (fileSigs.length!=ownSigs.size()) {
        throw new IllegalStateException(
          "Extension signatures do not match APK signatures");
      }

      for (Signature sig : fileSigs) {
        if (!ownSigs.contains(sig)) {
          throw new IllegalStateException(
            "Extension signatures do not match APK signatures");
        }
      }
    }
    catch (Exception e) {
      throw new IllegalStateException(
        "Could not validate extension APK", e);
    }
  }
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

Loading and Using an Extension

Next, we need to get our hands on the `QuestionsLoader`.

To that end, `ThingsLoaderThunk` holds onto another `ThingsLoader`, in an `extImpl` private field, as well as an single-thread `Executor` cunningly named `executor`.

The `ThingsLoaderThunk` constructor also sets up a few other fields of note:

```
ThingsLoaderThunk(Context ctxt, String url,
                  String classname, Callback cb)
  throws MalformedURLException,
  PackageManager.NameNotFoundException {
  this.classname=classname;
  this.cb=cb;

  pm=ctxt.getPackageManager();
```

```
    ownSigs=
      Arrays.asList(pm
        .getPackageInfo(ctxt.getPackageName(), PackageManager.GET_SIGNATURES)
        .signatures);

    this.url=new URL(BuildConfig.EXTENSION_URL);

    tmb=new TrustManagerBuilder().withConfig(ctxt,
      R.xml.extension_server);

    String basename=Uri.parse(url).getLastPathSegment();

    apkPath=new File(ctxt.getCacheDir(), basename);
    cachePath=new File(ctxt.getCacheDir(),
      UUID.randomUUID().toString());

    cachePath.mkdirs();
  }
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

Specifically, we have:

- `classname`: the Java class name for the `ThingsLoader` implementation in the extension
- `apkPath`: the local copy of the APK that we downloaded
- `cachePath`: an empty unique directory for Android to cache data unpacked from the APK file

We also hold onto `cb`, the `ThingsLoaderThunk.Callback` implementation supplied by `ThingsFragment`:

```
interface Callback {
  void onError(String message, Exception e);
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

The `ThingsLoaderThunk` `startAsyncLoad()` method glue a lot of this together:

```
@Override
public void startAsyncLoad() {
  if (extImpl==null) {
    executor.execute(new Runnable() {
      @Override
      public void run() {
```

```
        if (!apkPath.exists()) {
          try {
            downloadExtension();
          }
          catch (Exception e) {
            reset();

            if (cb!=null) {
              cb.onError("exception in HTTP", e);
            }
          }
        }

        try {
          loadThunk();
          extImpl.startAsyncLoad();
        }
        catch (Exception e) {
          reset();

          if (cb!=null) {
            cb.onError("exception in loadThunk()/startAsyncLoad()",
              e);
          }
        }
      }
    });
  }
  else {
    extImpl.startAsyncLoad();
  }
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

The simplest scenario is if `extImpl` is not `null`, meaning that we used this thunk previously. In that case, `startAsyncLoad()` just forwards the call along to the `extImpl` `ThingsLoader` implementation.

However, the first time we use the thunk and call `startAsyncLoad()`, `extImpl` will be `null`. In that case, on our single-thread `Executor`, we see if we already have the APK file downloaded. If we do not, we call `downloadExtension()`, to download and validate the APK. If there is a problem, among other things, we call `onError()` on the `Callback`, to pass the exception up to the `ThingsFragment`, to display to the user.

Once the APK is downloaded, we then call `loadThunk()`:

```
private void loadThunk()
  throws ClassNotFoundException, IllegalAccessException,
  InstantiationException {
  DexClassLoader dcl=
    new DexClassLoader(apkPath.getAbsolutePath(),
      cachePath.getAbsolutePath(), null,
      getClass().getClassLoader());
  Class<ThingsLoader> clazz=
    (Class<ThingsLoader>)dcl.loadClass(classname);

  extImpl=clazz.newInstance();
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

The guts of the whole dynamic-code engine in Android is `DexClassLoader` and related classes. `DexClassLoader` wraps around an APK (or a JAR file containing a `classes.dex` file) and gives you a Java `ClassLoader`, for use in accessing classes implemented in the APK or JAR file.

The `DexClassLoader` constructor takes four parameters:

- A filesystem path to the APK or JAR (or a list of such files, separated by colons)
- A filesystem path to a directory that the framework can use for caching information that gets unpacked or converted from the APK or JAR file(s)
- A colon-delimited list of paths to native libraries used by this code (which will be `null` unless you have tremendous tolerance for debugging pain)
- The parent class loader, for use in resolving classes not defined locally within the library (typically obtained by calling `getClassLoader()` on some existing `Class` object)

So, `loadThunk()` creates the `DexClassLoader`, then uses standard Java `ClassLoader` behaviors to load our `ThingsLoader` class based on its fully-qualified class name, then creating an instance of that class using the zero-argument constructor (courtesy of `newInstance()` on the Java `Class` object).

Finally, after having loaded the thunk, we forward the `startAsyncLoad()` call to it.

Given our extension implementation and API, this will cause a `ThingsLoadedEvent` to be raised, which our `ThingsFragment` picks up. `ThingsFragment` then calls `getThings()` on the thunk, which just forwards the call along to the extension:

```
@Override
public List<Thing> getThings() {
  if (extImpl==null) {
    return(null);
  }

  return(extImpl.getThings());
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

`ThingsFragment` then just wraps the `List` of `Thing` object in a `ThingsAdapter`, putting that into the `ListView`, using code very similar to the original Stack Overflow examples.

The net effect is that when you run the host app, it downloads the extension APK, validates it, gets an instance of the `QuestionsLoader`, and uses that to populate the `ListView`.

Resetting the Environment

We need to clear out all this stuff on occasion. That is handled by a `reset()` method on `ThingsLoaderThunk`:

```
void reset() {
  extImpl=null;
  ZipUtils.delete(apkPath);
  ZipUtils.delete(cachePath);
}
```

(from DynCode/ListLoader/app/src/main/java/com/commonsware/android/dyncode/ThingsLoaderThunk.java)

Here we `null` out `extImpl` (as it will shortly be no longer safe to use), then wipe out the locally-stored materials from `apkPath` and `cachePath`. `ZipUtils.delete()`, from the CWAC-Security library, is simply a recursive-delete implementation, mostly for the `cachePath` directory's benefit.

One place that we call `reset()` is if something goes wrong when trying to download, validate, or use the APK. If there is an unhandled exception from `downloadExtension()` or `loadThunk()`, we call `reset()` to ensure we have a clean slate for future work.

Also, `ThingsFragment` has an action bar item, "Reset", which triggers a call to `reset()`. This allows the user to request that we wipe out all this information. Along

the way, clicking “Reset” also exits the activity, since we no longer have our extension classes.

You could do similar sorts of cleanup in other cases:

- Downloading an updated APK
- As part of users disabling certain extensions through some sort of settings interface
- As part of `onDestroy()` processing in an activity or fragment, if `isFinishing()` returns `true`, indicating that the activity instance is no longer needed, if you want to force a fresh extension download for future activity instances

What is impractical is attempting to clear this stuff out when your process terminates. `onDestroy()` on your activity or service may not be called, and there is no guaranteed way of getting control when your process is being terminated by the system.

What You Need To Run This Sample

This sample app will need some customization if you wish to try to run it on your development machine.

First, decide where you want the extension APK to be downloaded from. Add a `gradle.properties` file in the overall project root directory, and in there have an `EXTENSION_URL` property, whose value is the URL that should be used to download the extension APK from:

```
EXTENSION_URL=https://your.server.goes.here/ext-debug.apk
```

If this is an HTTPS URL, you will need to follow the guidance in the chapter on SSL to change `app/main/res/xml/extension_server.xml` with the appropriate rules for how to validate the SSL certificate from your server or otherwise minimize the risk of MITM attacks.

If this is a plain HTTP URL, please switch to an HTTPS-capable server.

If you insist upon using a plain HTTP URL, despite the clear security risks in doing so, comment out all the `TrustManagerBuilder` statements in `ThingsLoaderThunk`.

Finally, build the extension APK, upload it to the designated location, then run the host app.

Note that with Android Studio 2.2+ (tested with Instant Run disabled), you can actually debug from the host app into the extension, by putting breakpoints in `ThingsLoaderThunk` and stepping from there into the extension code. This assumes that your extension APK is a `debug` build, of course.

Additional Concerns

The `ListLoader` sample project demonstrates the mechanics of setting up dynamic loading of code compiled to DEX. However, it does not illustrate all the possible issues or variations on the theme that may interest you. Here are some additional concerns that you may have and points on how they tie into the sample.

Storage Locations

The sample app uses `getCacheDir()` for two roles:

- storing the downloaded APK that serves as the extension
- storing optimized code and other things created by the framework as part of setting up the `DexClassLoader`

`DexClassLoader` expects both of these to be filesystem paths. You cannot use a `content` `Uri` obtained from the Storage Access Framework, for example. If you wanted to allow the user to choose an extension APK via `ACTION_OPEN_DOCUMENT`, you would need to do that as some sort of "import" operation, making a local copy of the APK at a filesystem location that you control.

`getCacheDir()` is the most likely candidate filesystem location for the sample app, as it is part of internal storage and therefore protected from access by arbitrary other apps (except for users with rooted devices). Using internal storage is almost assuredly the right answer for the filesystem paths to use with `DexClassLoader`.

If, for some crazy reason, you want to use other locations, standard Android access rules apply:

- External storage may require the `WRITE_EXTERNAL_STORAGE` permission, depending upon where you want to store the material and the version of Android that your app is running on

- Removable storage is limited to the locations returned by `getExternalFilesDirs()` and `getExternalCacheDirs()`, and that only on Android 4.4+

Detecting Extension Updates

The sample app does not attempt to detect whether or not the APK was replaced with some newer edition. As a result, the only way to "update" the extension is to use the Reset action bar item to wipe out the existing downloaded extension copy, so that on the next launch, the app is forced to download the extension again.

However, typically, if you are bothering with all of this, you intend to ship updates to the extensions, more frequently than you ship updates to the main app APK. How you ship those updates depends a lot on how you ship the extensions in the first place.

If you go with an approach reminiscent of the sample app — downloading extensions from a Web server — you have a few options for delivering updates:

- Use standard HTTP caching protocols (e.g., `ETag`, `If-Modified-Since`). This is simplified if you use an HTTP client library that helps with the caching, such as OkHttp.
- Instead of downloading the extension directly, you first retrieve some metadata about the extension, such as version information and the URL for the latest version of the extension. This will allow you to hold onto version information about your current extension, to compare with the latest one, to see if your extension is up to date. This is a bit reminiscent of how the EmPubLite tutorial handles getting updates to the book content, for example.
- Use push messaging (e.g., FCM) to tell clients about updated extensions, using one of the preceding approaches as a fallback.

If, however, you are distributing the extensions by other means, you will need to figure out an appropriate update mechanism.

Accessing Resources and Assets

There are no documented ways to get access to resources from an external APK file. If it were installed, you could use `createPackageContext()` to access its resources, but that is not an option for non-installed APKs.

For assets, while there is no seamless access to them, you should be able to use the existing ZIP classes in the Android SDK (e.g., `ZipFile`), as an APK is merely a ZIP archive.

ProGuard Settings

In many cases, your extension will not use its own API. As a result, in a `release` build, ProGuard may attempt to either remove the seemingly-unused classes or possibly "merely" rename them as part of obfuscating your code. Either of these will prevent the host app from using the extension, as either the desired API classes are missing or are not there under their regular names.

You will want to add appropriate ProGuard `-keep` rules to block this, typically in a `proguard-rules.pro` file in the project root directory. Alternatively, add the `@Keep` annotation to your API classes.

Adding Permissions

Since the extension is packaged as an APK, you might think that you can have `<uses-permission>` elements in the extension's manifest, and that those might take effect when the extension is loaded.

Alas, no. The manifest entries only get used if the APK is installed, and in this case, we are not installing the extension APK. As a result, extensions are limited to whatever permissions are held by the host app and cannot add new ones.

If your `minSdkVersion` is 23 or higher, you could consider having the host app's manifest contain a lot more `<uses-permission>` elements than you need initially, just in case you decide that an extension should use Android SDK features that require those permissions. Your extension API can request Android 6.0+ runtime permissions, just as your host app can — the extension simply cannot add new `<uses-permission>` elements to the "combined" app.

However, bear in mind that requesting lots of unnecessary permissions in the host app may concern some users who closely inspect what permissions an app asks for. It also increases the potential for danger if, despite your best efforts, somebody successfully replaces your extension with one laden with malware, as now that malware can perform whatever your permissions allow.

Adding Components

Extensions cannot add new activities, services, or providers, for much the same reason as why it they cannot add new permissions — they cannot modify the host app's manifest. Extensions can call `registerReceiver()` to register a `BroadcastReceiver`, assuming that they get passed in a suitable `Context`, but extensions cannot register receivers in the host app's manifest.

Your host app could implement some generic components, though. For example, your host app could have an `ExtensionActivity` that extends `Activity` and forwards select method calls along to an extension-supplied class. That class cannot be an `Activity` itself, as you have no good way of properly initializing the `Context` that forms the foundation of the `Activity`. However, that does not stop you from "filling in the blanks" of the host app's `ExtensionActivity` with custom behavior. You could even use data in the `Intent` used to start `ExtensionActivity` (e.g., action string) to determine which extension-supplied class should be used for a given `ExtensionActivity` instance. And, the same technique should work for `Service`, `ContentProvider`, and `BroadcastReceiver`.

However, there are limitations:

- The extension cannot change any component attributes defined in the manifest, such as `android:label`
- General Android limitations still apply, such as only having one instance of a given `Service` at a time, which may constrain your flexibility in places

Having Several Extensions

The sample app demonstrates having one extension. Other than standard limitations like available heap space, there are no obvious limits as to how many `DexClassLoader` instances you can have, and therefore how many different extensions you can load. This includes:

- Having multiple extensions implementing the same API, such as several distinct `ThingsLoader` extensions to supply `Thing` instances from various data sources
- Having multiple extension APIs, with zero or more extensions for each

Of course, the more extensions (and APIs) you juggle, the greater the complexity of your app and the greater the chance of bugs.

Changing the Extension API

Where things really start to come apart, though, is if you wish to change the extension API.

Extensions and the host app share the common API definition, but that assumes an unchanging API. If the host app were to use a newer API than does the extension, you may encounter `VerifyErrors` and related crashes when the host app tries to invoke non-existent methods in the extension-supplied implementation.

An ideal case is to update the host app and all extensions at one time. You could use some metadata in the extension distribution process indicate what API version the extension supports. Old editions of the host app would only update to compatible extensions; new editions of the host app would download the latest extensions.

If updating the host app and its extensions simultaneously is impractical, you can attempt to version the APIs themselves. For example, `ThingsLoader` might be the original interface, where `ThingsLoaderV2` might represent a newer edition of the interface. Once again, version metadata would indicate to the host app what API versions the extension supports, so older host apps could stick to the API version they know about while newer host apps could instantiate classes implementing the newer interface.

Is Any Of This a Good Idea?

If you are distributing your apps privately to a limited set of users, and you feel comfortable that you can handle the security and complexity issues, using the techniques in this chapter — particularly the `DexClassLoader` — is not completely insane.

If you are distributing your apps publicly to a large audience, but you have a large firm that is flush with cash and can spend lots of money ensuring that you are handling the security properly... using these techniques is not a great idea, but at least it is one that other similar firms have applied.

Otherwise, be very wary of dynamic code updates.

Android Studio Editors and Dialogs

Eclipse, with the ADT plugin, had many structured editors and specialized dialogs for modifying Android project files and otherwise configuring Android project behavior.

Android Studio has fewer of those, and they are generally less critical. The editors and dialogs presented in this chapter can be useful, at least in some cases, but you do not need to use any of them to be able to create your Android projects. However, some may speed up your Android development a bit over working with bare resource and Gradle files.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, along with the chapter on Gradle and build variants.

Project Structure

The Project Structure dialog allows you to configure many aspects of your `build.gradle` files from a tabbed property-style dialog, as opposed to having to work with the Gradle scripts directly. On the plus side, this can be easier. However, since Gradle is built on the Groovy scripting language, `build.gradle` files are not simple XML or JSON data structures. It remains to be seen how well the Project Structure dialog will be able to handle complex Gradle scripts.

To access the Project Structure dialog, choose File > Project Structure from the main IDE menu.

The left-hand side lists major areas of the dialog; choosing one of those switches to that area's form on the right.

The sections that follow outline each of the major areas and what you can configure in them.

SDK Location

The Project Structure dialog opens up on the SDK Location area, where you can configure where your Android SDK is located, where your JDK is located, and where your NDK is located:

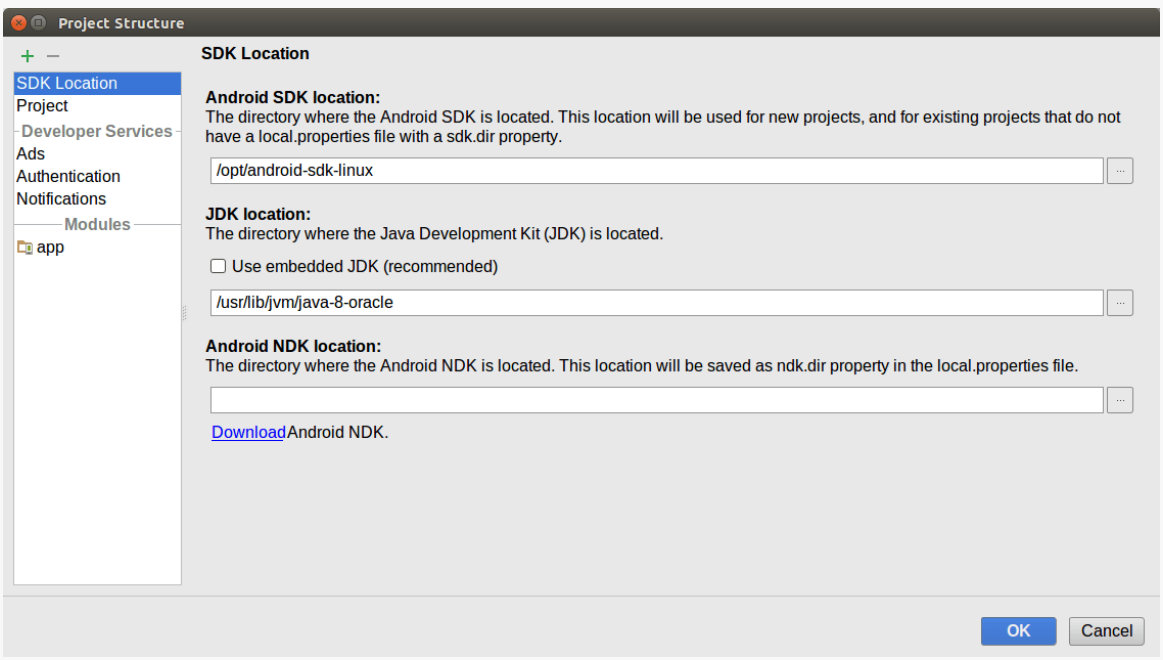


Figure 919: Project Structure Dialog, SDK Location Category

For new Android Studio 2.2+ installations, the default is for Android Studio to use a version of the Java JDK that ships with the IDE itself, in which case "Use embedded JDK (recommended)" will be checked.

Adjusting these in Project Settings affects this specific project. There is also File > Other Settings > Default Project Structure, where you can edit the default values to be used for new projects and projects that you import in the future.

Project Settings

The second entry in the Project Structure dialog category list is "Project". This allows you to configure four items found by default in the `build.gradle` file in your project root or in the `gradle-wrapper.properties` file:

- What version of Gradle you wish to use for the Gradle Wrapper
- What version of the Android Gradle Plugin you wish to use
- What artifact repository should be used for pulling in the Gradle for Android plugin (and any other plugins you may be using)
- What artifact repository should be used by default for standard module artifacts (e.g., those you request via `implementation` directives in your module's `build.gradle` file)

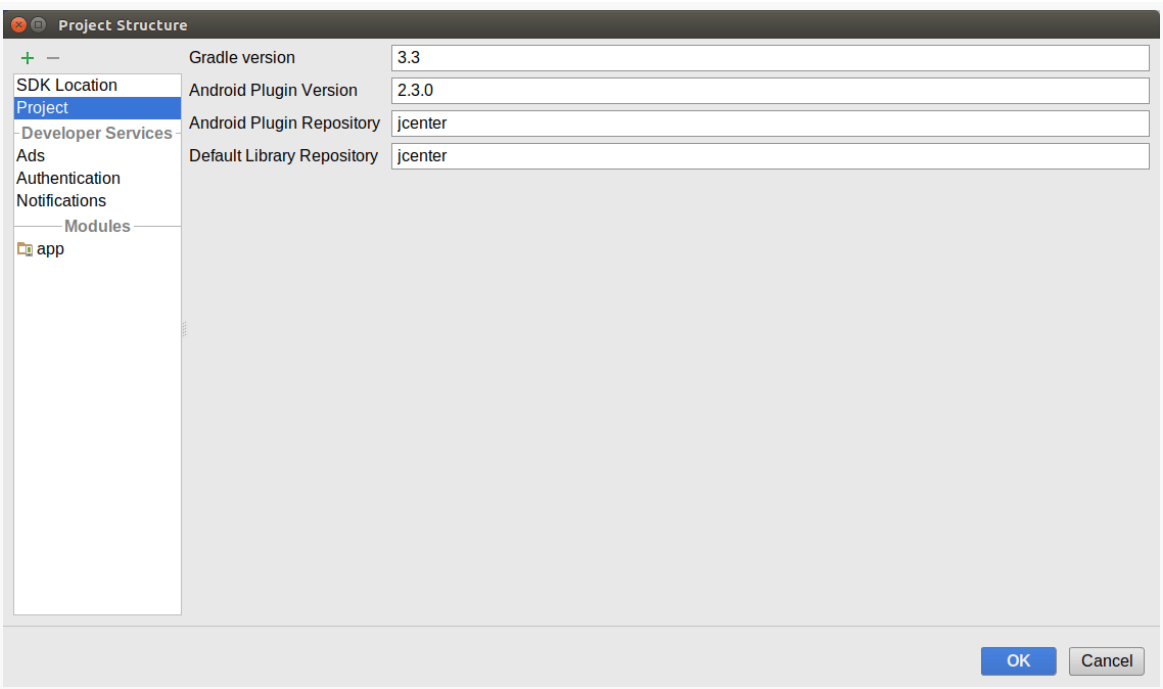


Figure 920: Project Structure Dialog, Project Settings Category

Developer Services

If you are using select portions of the Play Services SDK, the items under the "Developer Services" divider allow you to configure those portions. By default, they amount to checkboxes, to enable certain features:

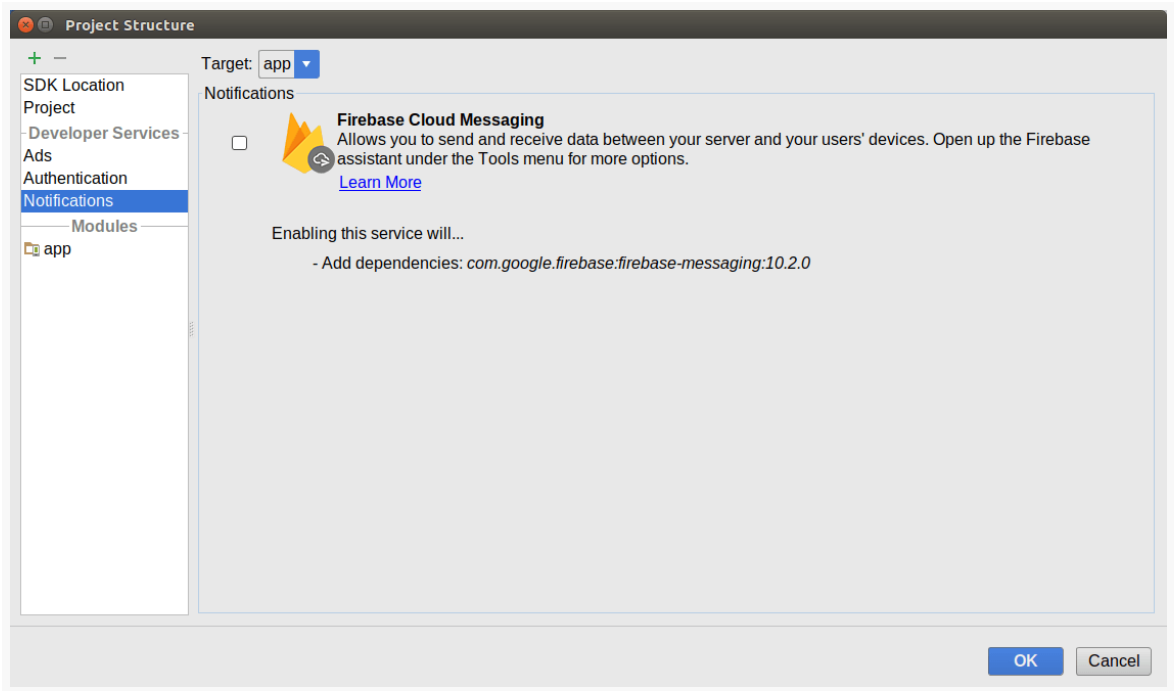


Figure 921: Project Structure Dialog, Notifications Category

Module Settings

Below the "Modules" divider in the category list on the left will come all of your modules. If you are not using modules, there will be a single entry in the category list with the same name as your project, as a quasi-module.

Clicking on a module will bring up a set of tabs on the right to edit various properties of that module, independently of any other module in your project. The following sections outline the contents of those tabs.

Properties

The first tab is labeled "Properties" and allows you to adjust various top-level settings in your module's `build.gradle` file.

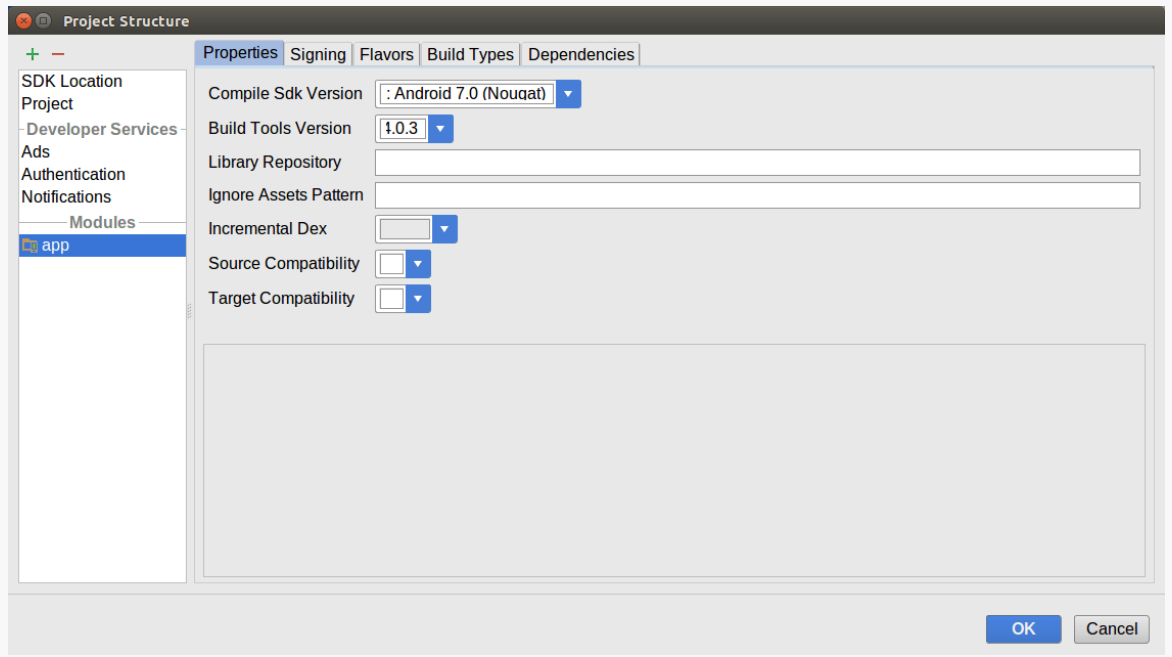


Figure 922: Project Structure Dialog, Module Category, Properties Tab

These include:

- Your `compileSdkVersion` ("Compile Sdk Version" drop-down)
- Your `buildToolsVersion` ("Build Tools Version" drop-down)
- Another artifact repository to use for this module, added to your module's `repositories` closure ("Library Repository")
- The `ignoreAssetsPattern` property in `aaptOptions` ("Ignore Assets Pattern")
- The `incremental` property in `dexOptions` ("Incremental Dex")
- The `sourceCompatibility` in `compileOptions` ("Source Compatibility")
- The `targetCompatibility` in `compileOptions` ("Target Compatibility")

Signing

If your module's `build.gradle` file has a `signingConfigs` closure, the "Signing" tab will let you edit those signing configurations:

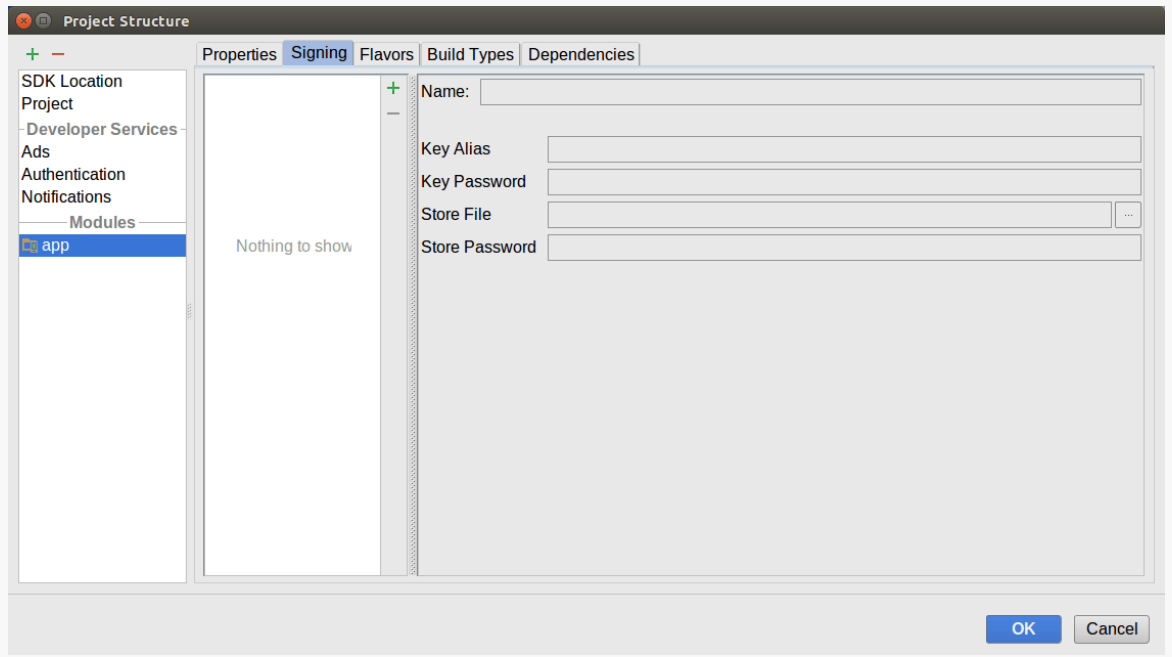


Figure 923: Project Structure Dialog, Module Category, Signing Tab

Each signing configuration that you have defined will appear in the list on the left side of the tab. On the right are fields for you to fill in the signing configuration name, the keystore file and key alias to use, and the passwords to use for accessing that file and alias.

The green plus ("+") icon on the right side of the list lets you define a new signing configuration, while the red minus ("-") icon lets you delete an existing signing configuration.

Flavors

The "Flavors" tab starts off with a single "flavor", representing your `build.gradle` file's `defaultConfig` settings. The green plus icon next to the list of flavors lets you define a new flavor, while the red minus icon lets you remove an existing flavor. Note that you cannot remove `defaultConfig`, as it is defined by the Gradle for Android plugin.

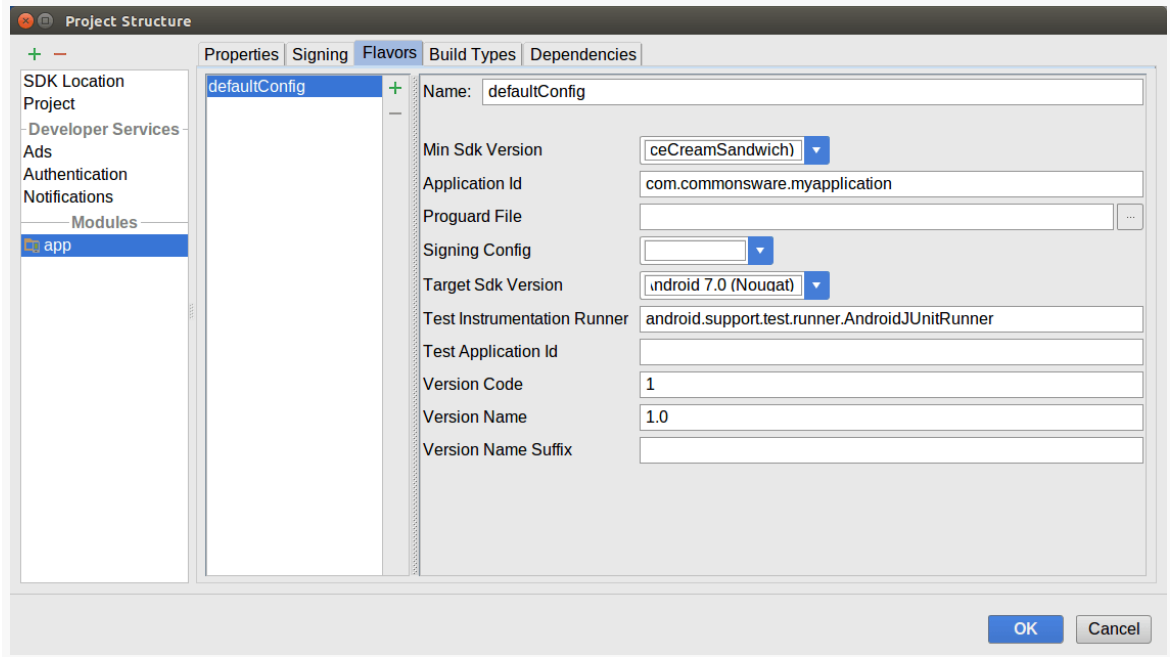


Figure 924: Project Structure Dialog, Module Category, Flavors Tab

On the right side of the tab, you can set or change the name of the flavor, plus you can adjust various flavor (or `defaultConfig`) settings, including:

- the `minSdkVersion` value ("Min Sdk Version" drop-down)
- the `applicationId` ("Application Id")
- the ProGuard rules file to use for builds ("Proguard File")
- which of your defined signing configurations to use ("Signing Config" drop-down)
- the `targetSdkVersion` value ("Target Sdk Version" drop-down)
- the `testInstrumentationRunner` to use for instrumentation testing ("Test Instrumentation Runner")
- the `testApplicationId` value for instrumentation testing ("Test Application Id")
- the `versionCode` and `versionName` to use ("Version Code" and "Version Name"), along with the suffix to apply to the version name (unique to this product flavor)

Build Types

The "Build Types" tab allows you to adjust settings for the `debug` and `release` build types. The green plus icon next to the list of build types lets you define a new build type, while the red minus icon lets you remove an existing build type. Note that you cannot remove `debug` or `release`, as they are defined by Gradle.

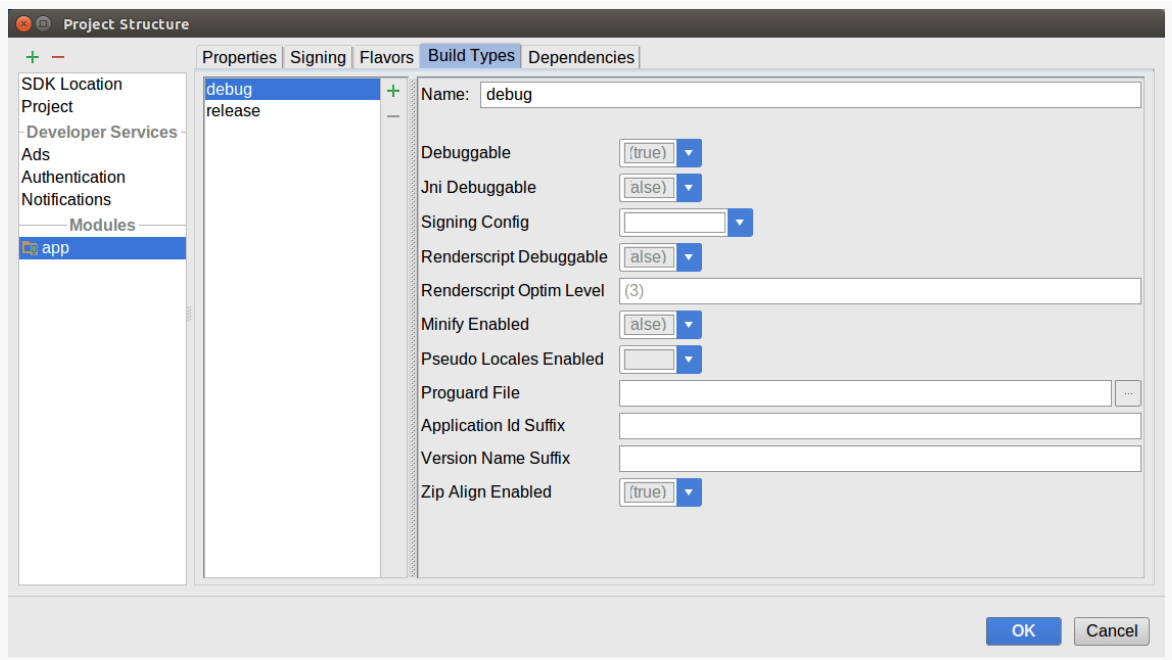


Figure 925: Project Structure Dialog, Module Category, Build Types Tab

On the right side of the tab, you can set or change the name of the build type, plus you can adjust various settings in your `buildTypes` closure, including:

- the value of `debuggable`, to control if the app is considered to be debuggable on production hardware ("Debuggable" drop-down)
- the value of the undocumented `jniDebuggable` flag ("Jni Debuggable")
- which signing configuration to use ("Signing Config" drop-down)
- the value of the undocumented `renderscriptDebuggable` flag ("Renderscript Debuggable")
- the value of the undocumented `renderscriptOptimLevel` property ("Renderscript Optim Level")
- the value of `minifyEnabled`, to control whether the build process should attempt to strip out unused code ("Minify Enabled" drop-down)

- the value of the undocumented `pseudoLocalesEnabled` flag ("Pseudo Locales Enabled")
- the ProGuard rules file to use for builds ("Proguard File")
- the suffix to append to the `applicationId` ("Application Id Suffix")
- the suffix to append to the `versionName` ("Version Name Suffix")
- whether the resulting APK should be processed by `zipalign` ("Zip Align Enabled")

Dependencies

If your project has any defined dependencies in a `dependencies` closure, these will appear in the "Dependencies" tab:

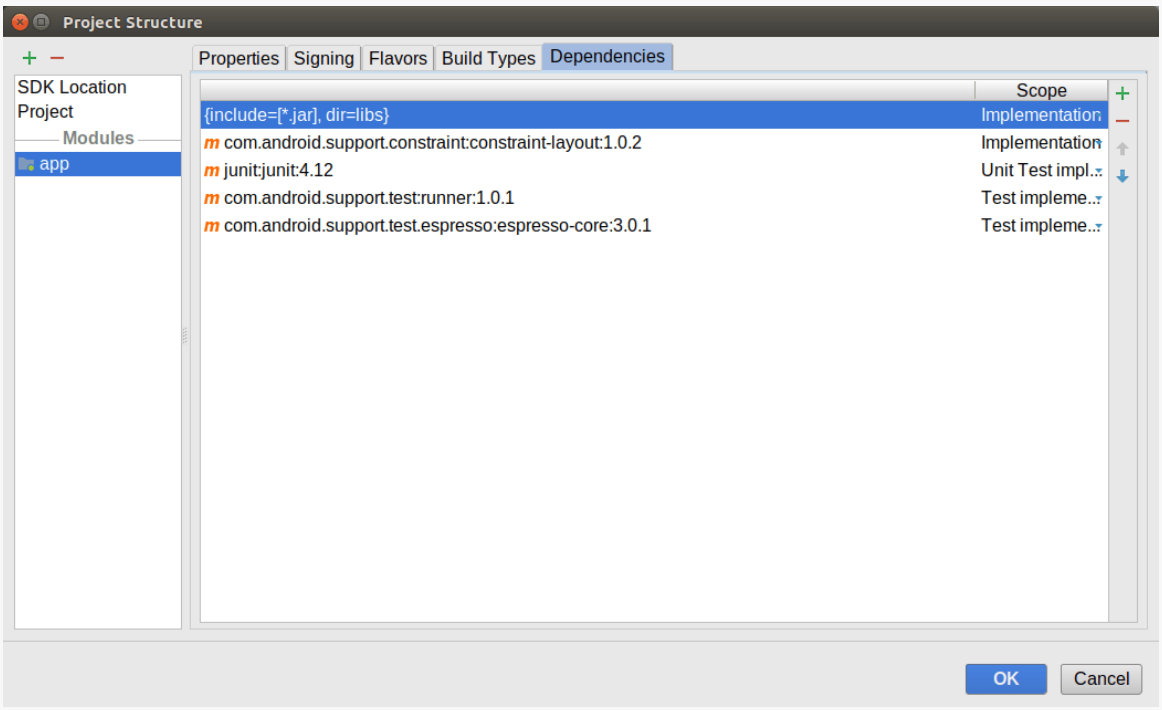


Figure 926: Project Structure Dialog, Module Category, Dependencies Tab

The tab is dominated by a two-column table, where the left column is the dependency itself. The right column is the "scope", where the cell shows the current scope, and if you click on it, you get a drop-down list of available scopes:

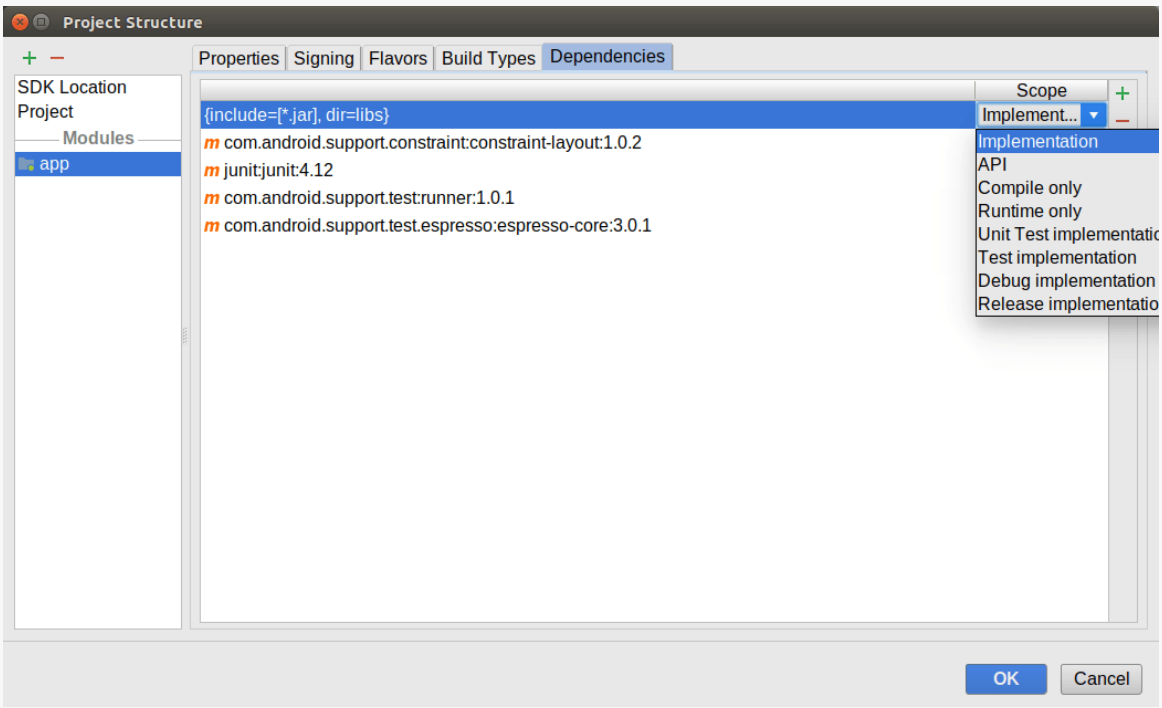


Figure 927: Dependencies Tab, Showing Scope Drop-Down

Those scopes include:

- "Implementation", for an `implementation` dependency
- "Test implementation" for an `androidTestImplementation` dependency (i.e., one to be used only for instrumentation testing)
- "Unit test implementation" for a `testImplementation` dependency (i.e., one to be used only for unit testing)
- Other "compile" scopes for your build variants (e.g., "Debug compile" for a `debugCompile` dependency)
- "Compile only", for a `compileOnly` dependency (where the dependency is used only at compile time and its contents are not packaged into the APK file)
- "Runtime only" for a `runtimeOnly` dependency (where the dependency is not used at compile time, but its contents are packaged into the APK file)

The latter two scopes will be used infrequently.

If you click the green + button, you will be able to add a new dependency. A drop-down menu will let you choose between a library dependency (i.e., for an artifact in

a repository), a file dependency, and a module dependency (i.e., to depend upon another module in your project).

Typically, you will be adding library dependencies. When you choose that option, another dialog appears to allow you to search for likely dependencies or type in the full dependency identifier (group ID:artifact ID:version).

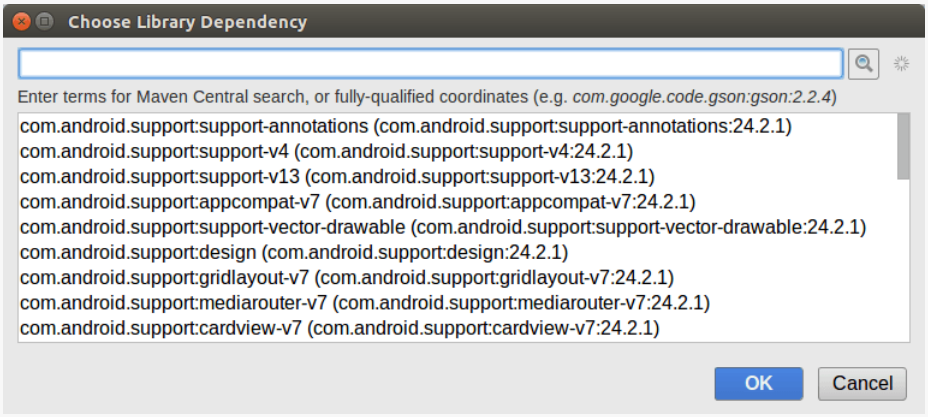


Figure 928: Choose Library Dependency Dialog, As Initially Launched

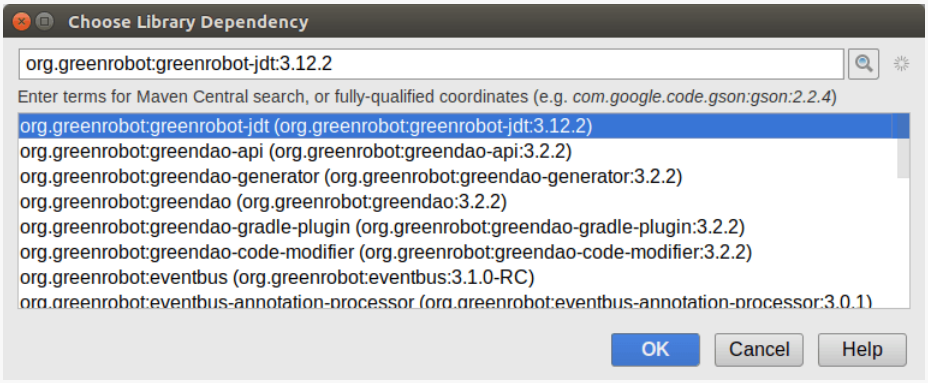


Figure 929: Choose Library Dependency Dialog, With Search Results for "greenrobot"

The red - icon in the same toolbar as the green + will remove a dependency, while the up and down arrows allow you to reorder the dependencies.

Translations Editor

On Android Studio, if you open a file containing string resources, you will find a notification banner atop the editor, offering a way for you to "Edit translations for all locales in the translations editor":

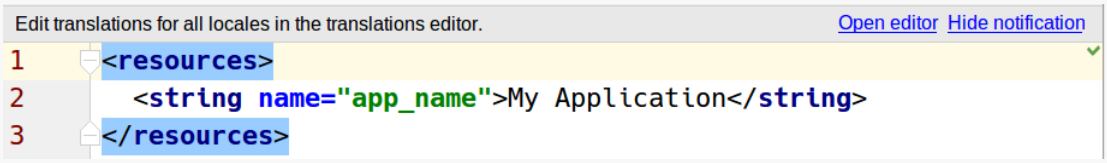


Figure 930: Notification Banner for Translations Editor

Clicking the "Open editor" link will open the Translations Editor. You can also get to this editor by right-clicking over the resource file in the Project or Android view on the left and choosing "Open Translation Editor" from the context menu.

For an un-translated project — such as one newly-created from the new-project wizard — when you open the Translations Editor, you will just see all of the existing strings:

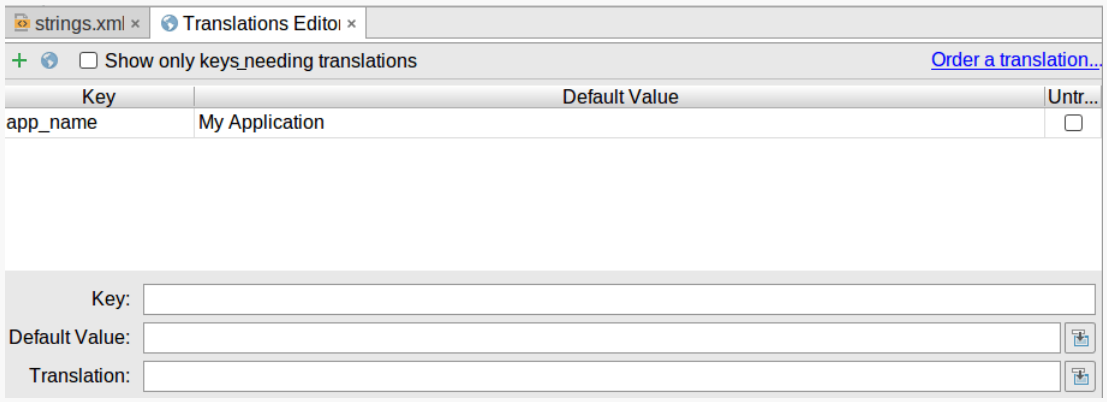


Figure 931: Translations Editor, As Initially Opened

These are labeled as "default value" because, in this case, the values are coming from the default resource set (`res/values/strings.xml`), not some specific language translation.

You can edit an existing default value either by clicking on the cell containing the default value (e.g., clicking the "My Application" cell), or by clicking anywhere on

the row and then editing the value in the "Default Value" field towards the bottom of the editor. Note that you cannot edit keys via this editor.

The right-hand column of the table has checkboxes, with a column heading of "Untranslatable". Checking one of those adds a `translatable="false"` attribute to the `<string>` element in the XML. The IDE and related tools can use this to not warn you that this string lacks translations. This would be good for strings that you elected to put in string resources yet are not user-facing and therefore do not need translation.

The + icon in the toolbar, when clicked, pops up a dialog where you can define a new string:

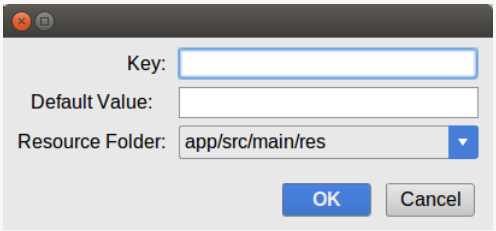


Figure 932: Translations Editor, New-String Dialog

Where the fun begins, though, is if you click the globe icon in the toolbar. This displays a drop-down list of languages:

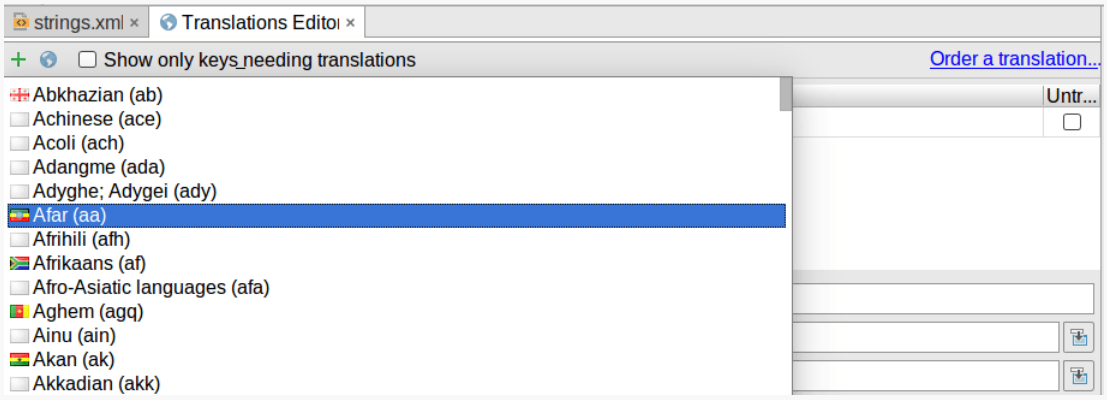


Figure 933: Translations Editor, Showing Languages Drop-Down List

Choosing a language has two main effects. First, it creates a corresponding `res/values-*/` directory for your chosen language. Second, it adds a column to the Translations Editor for that language:

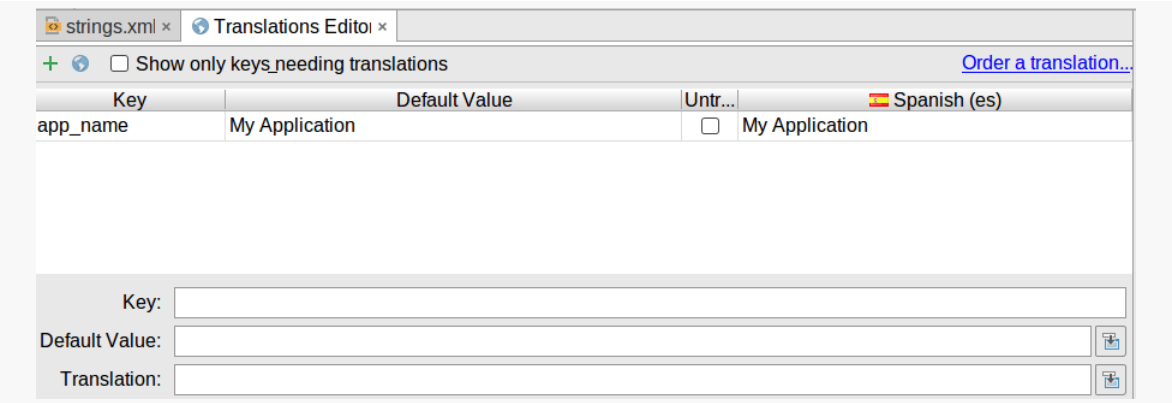


Figure 934: Spanish Strings in Resource File and Translations Editor

You can then click on a cell representing a word and its language, and fill in the translation in the form:

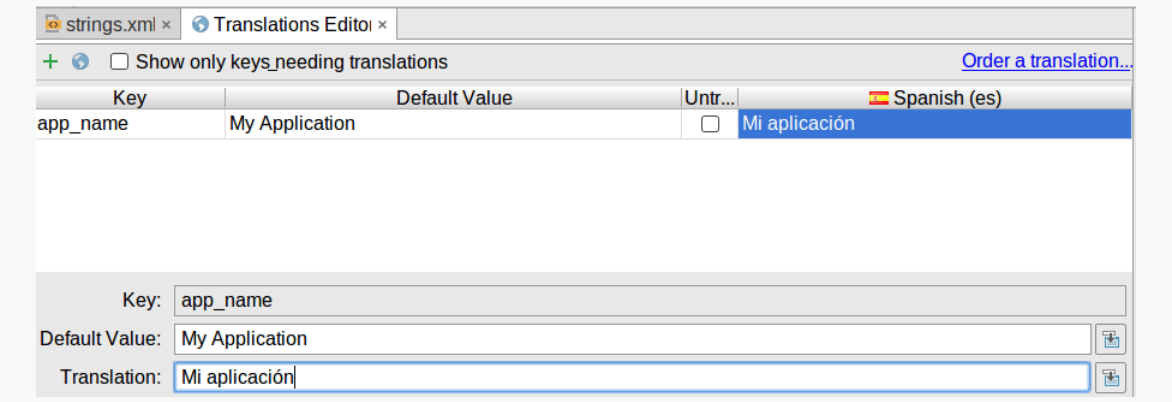


Figure 935: Translations Editor, Showing Spanish Translation

The icons to the right of the "Default Value" and "Translation" fields in the form simply pop up a dialog giving you a bit more room to type:

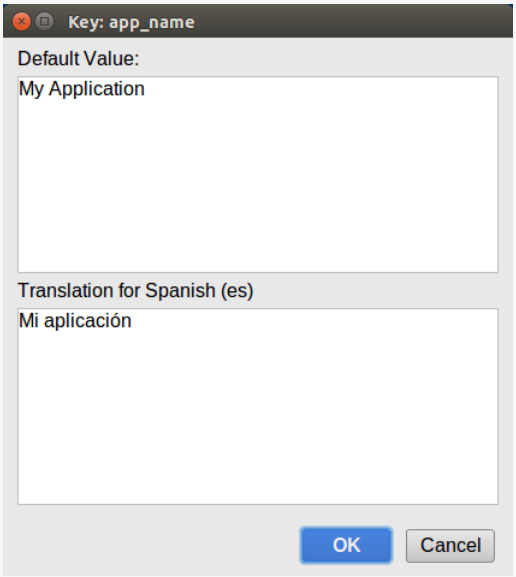


Figure 936: Translations Editor, Values Edit Dialog

Trail: Other Tools

Advanced Emulator Capabilities

The Android emulator, at its core, is not that complex. Once you have one or more Android virtual devices (AVDs) defined, using them is a matter of launching the emulator and installing your app upon it. With Android Studio, those two steps can even be combined — the IDE will automatically start an emulator instance if one is needed.

However, there is much more to the Android emulator. This chapter will explore various advanced features of the emulator and how you can use them.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Other Notable Configuration Options

When defining an AVD, or editing an existing AVD definition, there are many other configuration options at your disposal.

Hardware Graphics Acceleration

Another way to speed up the emulator is to have it use the graphic card or GPU of your development machine to accelerate the graphics rendering of the emulator window. By default, the emulator will use software-based rendering, without the GPU, which is slow in general and worse when running an ARM-based image.

Whether this will work or not for you will depend in part upon your graphics drivers of your development machine. Also, their use might conflict with other things you might want to do — on Linux, using hardware GPU mode might break your ability to take screenshots, for example.

This setting is toggled within the AVD Manager, for new and existing AVDs, via the "Graphics" drop-down list in the "Emulated Performance" group:

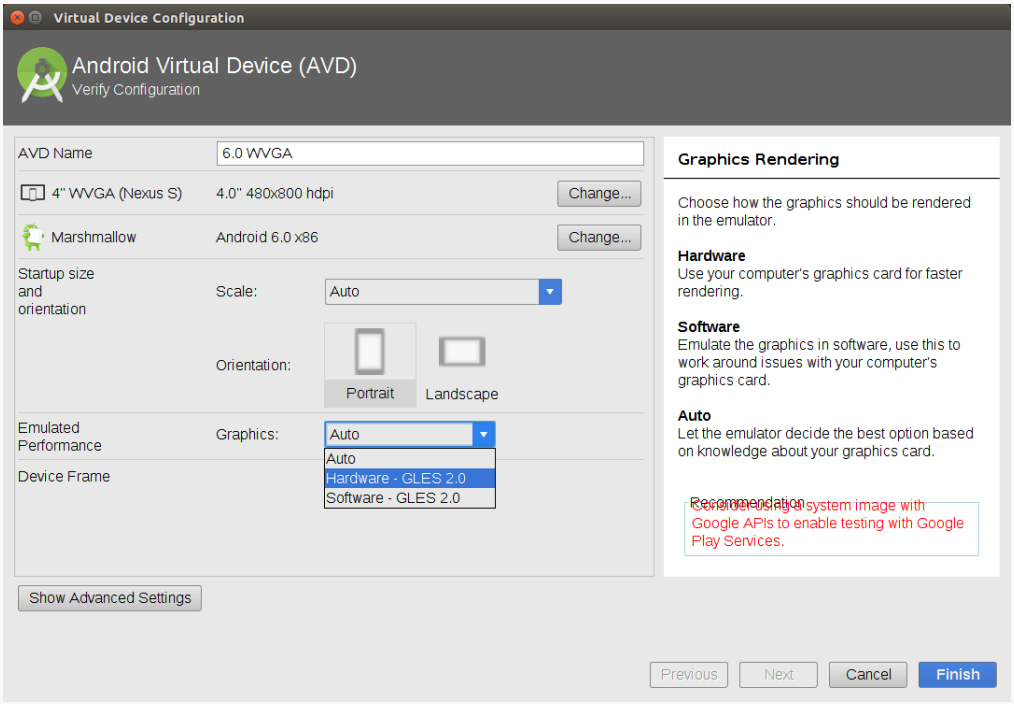


Figure 937: Virtual Device Configuration, Showing "Use Host GPU" Checkbox

There are three options:

- "Software" says to render the graphics purely within the emulator software
- "Hardware" says to render the graphics using the GPU of your development machine
- "Auto" (the default) delegates the decision to the emulator itself, based on its own heuristics of what will work well

Keyboard Behavior

The Android emulator can emulate devices that have, or do not have, a physical keyboard. Most Android devices do not have a physical keyboard, and so the

emulator is set up to behave the same. However, this means that typing on your development machine's keyboard will not work in `EditText` widgets and the like — you have to tap out what you want to type on the on-screen keyboard.

If you wish to switch your emulator to emulate a device with a physical keyboard – either "for realz" or just to simplify working with the emulator on your development machine — you can do so.

In the Android Studio AVD Manager, in the "Advanced Settings" area, there is an "Enable keyboard input" checkbox that determines whether hardware keyboard input is honored in the AVD or not:

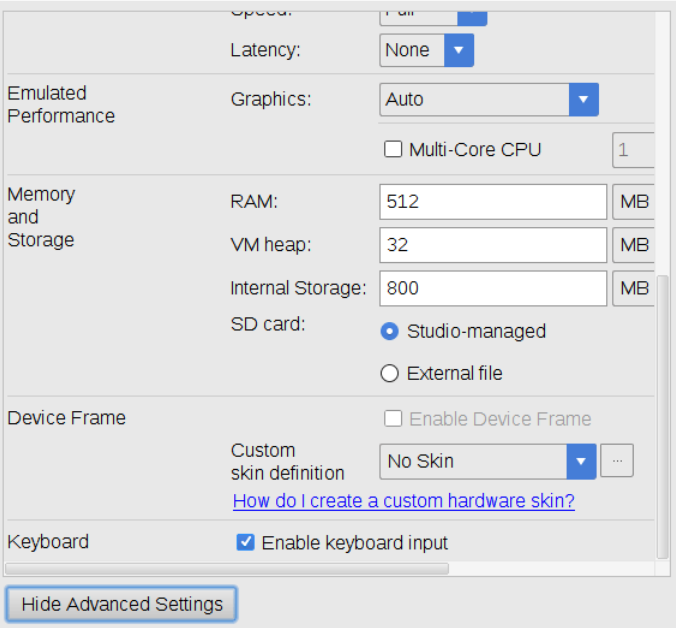


Figure 938: Virtual Device Configuration, Showing "Enable keyboard input" Checkbox

Startup Settings

Pixels on your development machine's monitor probably are substantially larger than the pixels on most Android devices. If the emulator tries to use one hardware pixel on your monitor for every emulated pixel of the device screen, your emulator may be bigger than your monitor can fit. The "Scale" drop-down controls how the emulator scales its output to deal with your monitor. "Auto" — the default value — probably is your best option, though you are welcome to use one of the other options to control the scaling more directly (e.g., `4dp` on the device maps to `1px` on your monitor):

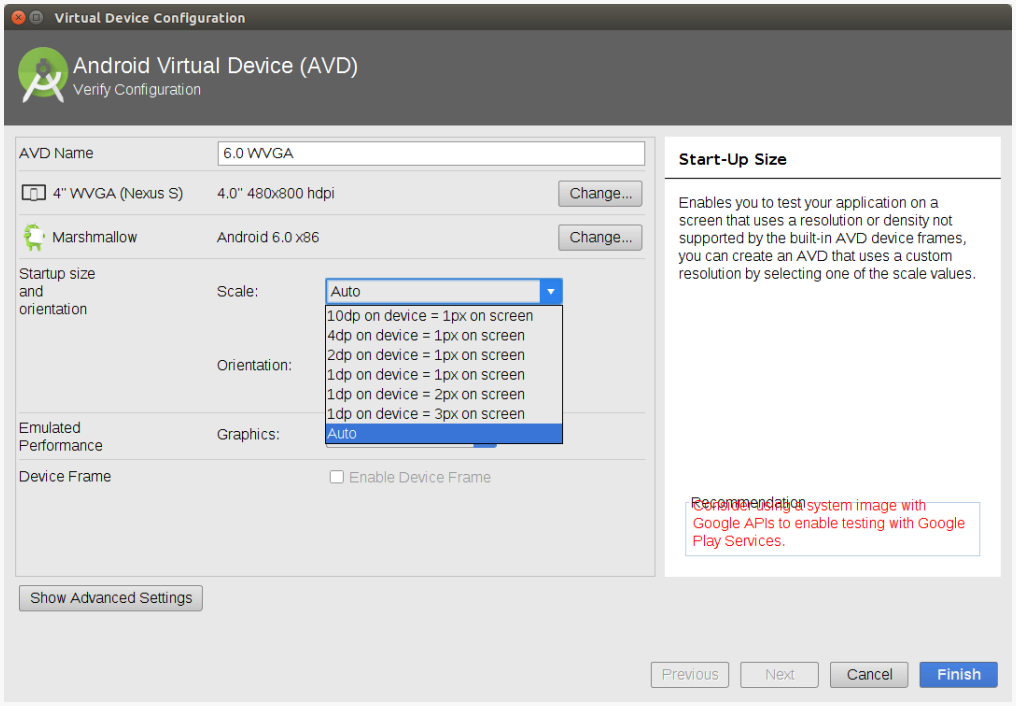


Figure 939: Virtual Device Configuration, Showing "Scale" Drop-Down

You can also control whether the device starts up in portrait or landscape mode at the outside, by the toggle buttons labeled "Orientation".

Note that scaling and orientation can also be controlled while the emulator is running; these settings merely control the startup conditions.

Camera Options

In the "Advanced Settings" area, you can control whether or not the emulator emulates a device with a camera:

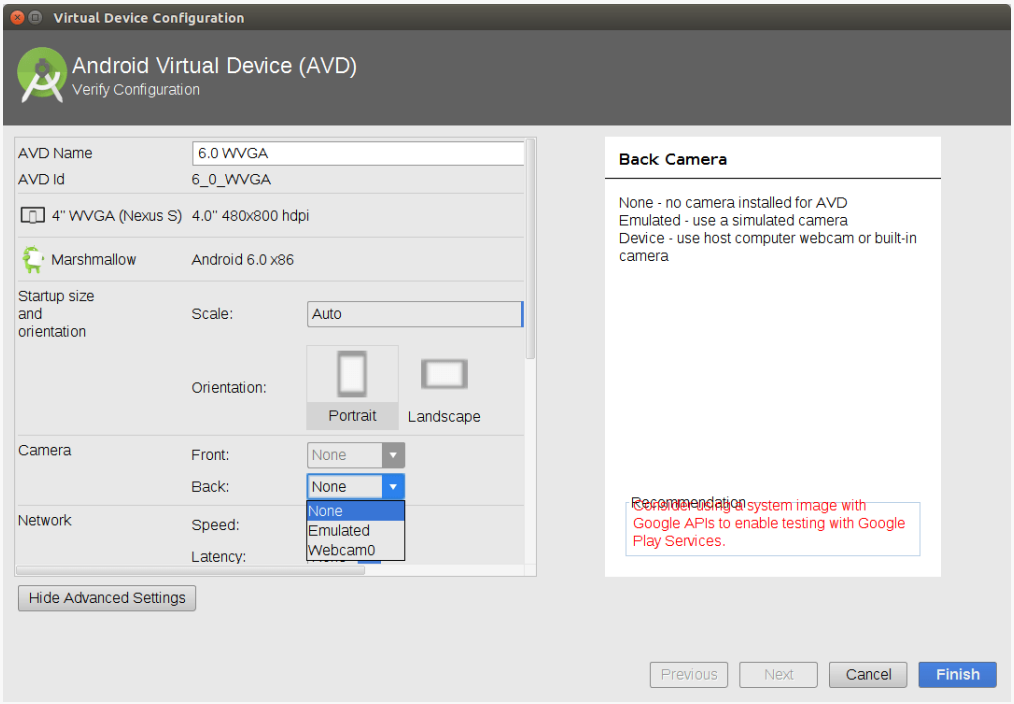


Figure 940: Virtual Device Configuration, Showing Camera Options

Whether you can configure both front and back cameras, or just one, is indeterminate. If you can configure a camera, your options are:

- "None", to emulate a device without a camera
- "Emulated", which emulates a device with a camera, but where the camera images themselves are emulated
- some hardware indicator (e.g., "Webcamo"), which emulates a device with a camera, where the camera images are pulled from some camera hardware on your development machine (e.g., a notebook webcam)

However, the emulator's ability to truly emulate the way Android cameras behave is very limited. Serious camera testing needs to be done using Android hardware, not the emulator.

Memory and Storage Configuration

In the "Advanced Settings" area, you can control how much RAM and storage is used by the emulator:

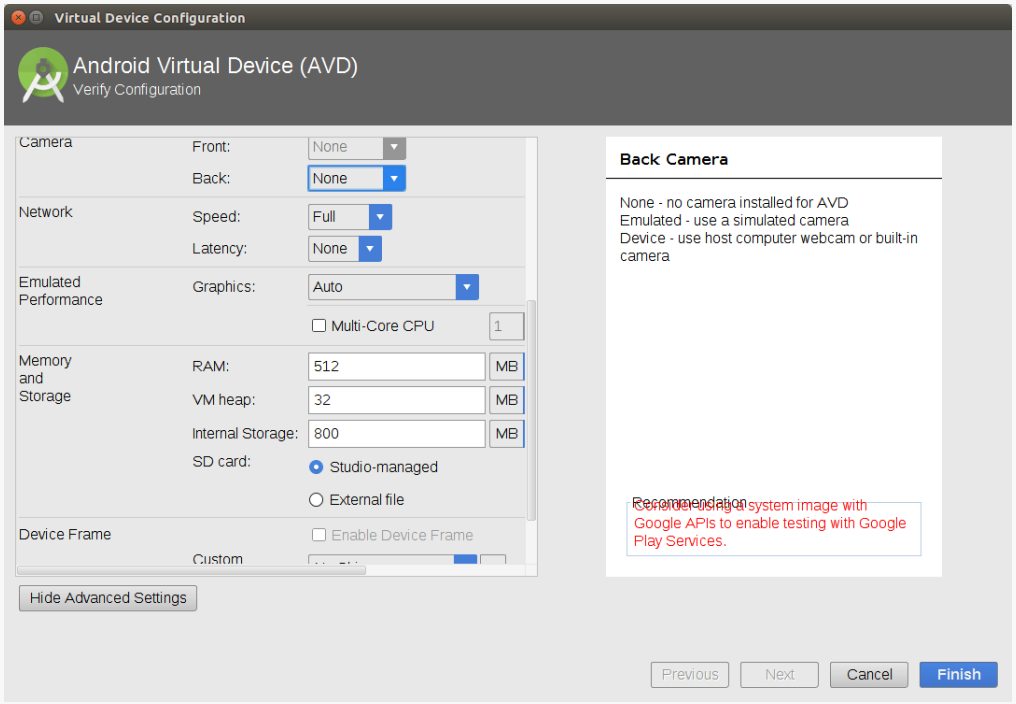


Figure 941: Virtual Device Configuration, Showing Memory and Storage Options

Specifically:

- "RAM" controls how much system RAM the emulator emulates. This will be a subset of the overall RAM of your development machine that the emulator consumes.
- "VM heap" appears to control the Dalvik/ART heap limit assigned to applications.
- "Internal Storage" indicates how much space is allocated for the main device partitions in the emulated device.
- "SD card" is still the misnomer for external storage. Your options are either to have Android Studio manage this for you, or for you to use tools like `mksdcard` to create your own disk image that you attach to the emulator.

Usually, the defaults are fine.

Frames and Skins

By default, the emulator appears in a bare window, showing the contents of the "touchscreen". Of course, an actual Android device will have more around it, such as bezels, optional hardware buttons, and so on.

In the “Device Frame” group in the “Advanced Settings” area, you can check “Enable Device Frame” and choose a skin to wrap around the touchscreen and make your emulator look a bit more like a real device:

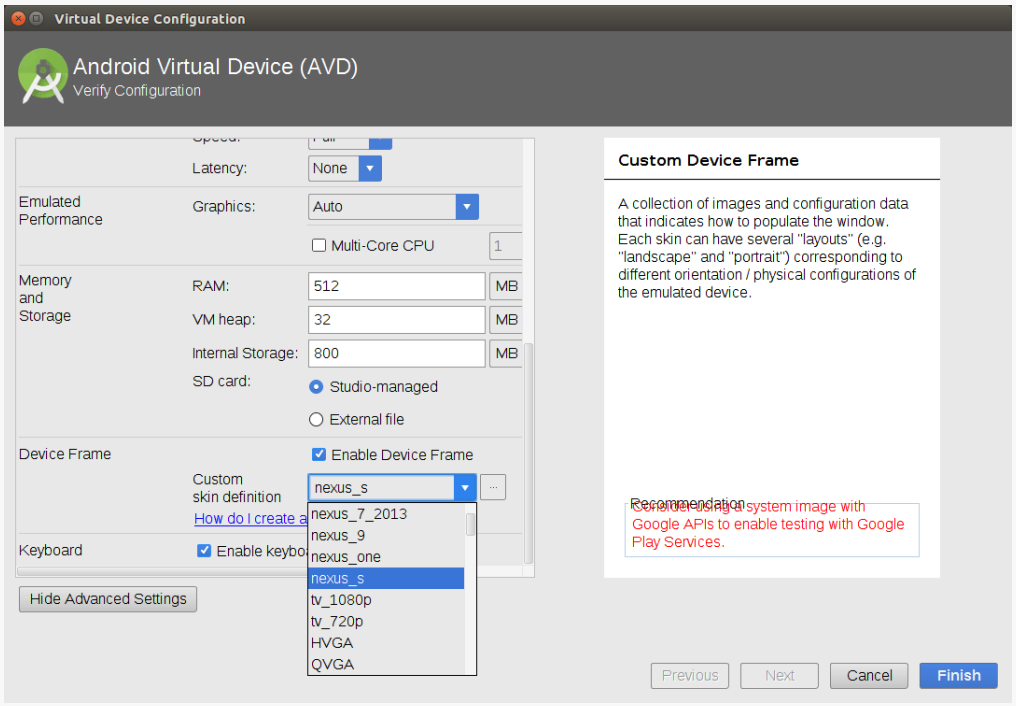


Figure 942: Virtual Device Configuration, Showing Device Frame Options

The Emulator Sidebar

Starting with Android Studio 2.0, the emulator sports a “sidebar” that runs alongside the main emulator window:

Figure 943: Android Emulator, with Sidebar on the Right

This provides you with rapid access to a number of emulator features and controls. Some of those are hidden behind the "More" button, at the bottom of the sidebar (looks like an ellipsis, "...").

Note that the sidebar buttons have tooltips that will tell both the button's purpose and the keyboard shortcut, if any, for that button.

Power and Navigation Controls

The top icon in the sidebar is a power button. A quick click on it will close your emulator. A long-click will behave like the POWER button on an Android device, bringing up the power menu:

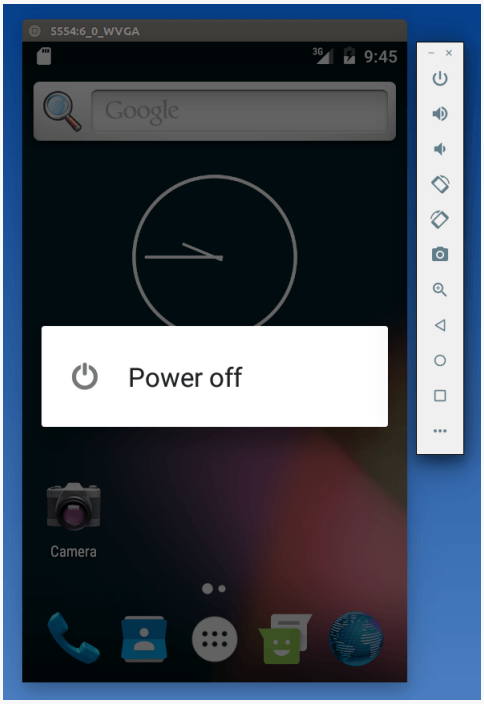


Figure 944: Android Emulator, Showing Power Menu

Towards the bottom of the sidebar are BACK, HOME, and RECENTS buttons for navigation:

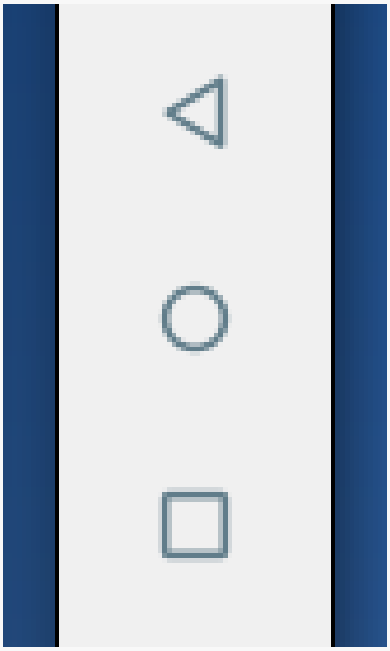


Figure 945: Android Emulator Sidebar Navigation Buttons

If you click the “More” button, you will open up the “Extended Controls” window:

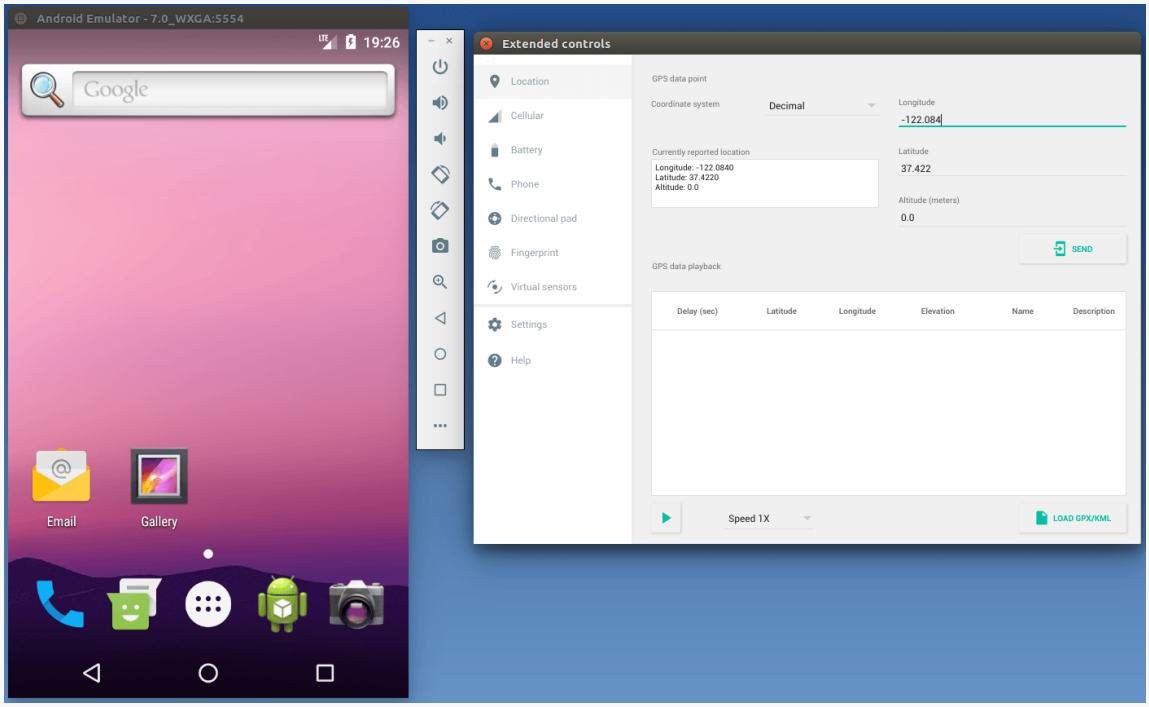


Figure 946: Android Emulator with Extended Controls

Clicking the “Directional pad” category on the left of the “Extended Controls” gives you D-pad and media buttons for in-app navigation:

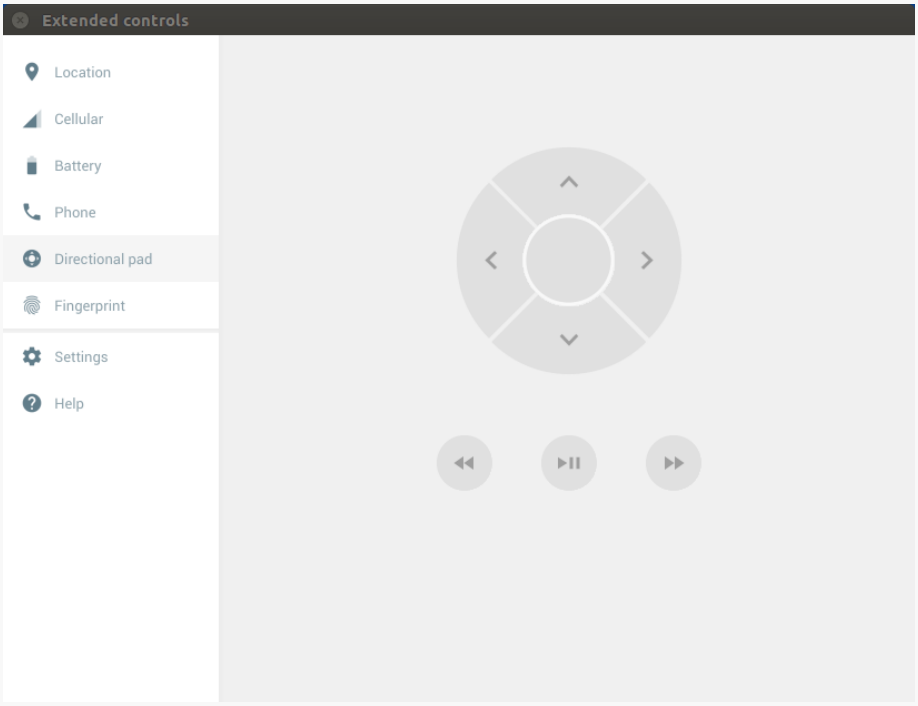


Figure 947: Emulator Extended Controls, Showing Directional Pad

Screen Orientation and Zoom

Two buttons on the sidebar allow you to rotate the device clockwise or counter-clockwise:

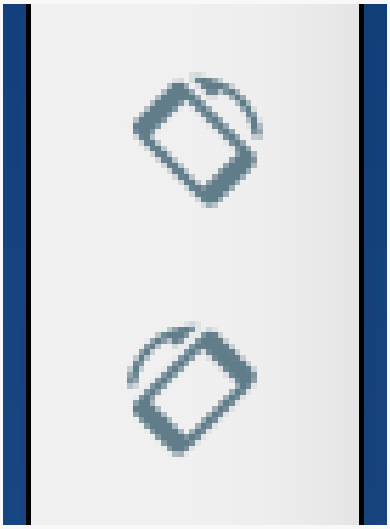


Figure 948: Android Emulator Sidebar Rotation Buttons

The magnifying glass icon allows you to zoom and out of the emulator screen contents:

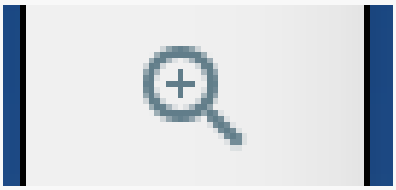


Figure 949: Android Emulator Sidebar Zoom Button

When in zoom mode, the mouse cursor changes to a magnifying class, and left-mouse clicks will zoom in at the clicked-upon point. Right-mouse clicks will zoom out. To return the mouse to normal behavior, tap the zoom sidebar button again. However, note that you will remain zoomed in on the last-selected zoom state; to return the emulator fully to normal, zoom out all the way first.

Screenshots

The camera button on the sidebar allows you to rapidly take screenshots of the emulator window:

Figure 950: Android Emulator Sidebar Screenshot Button

These will be stored in a directory controlled by the “Settings” category in the “Extended controls” window:

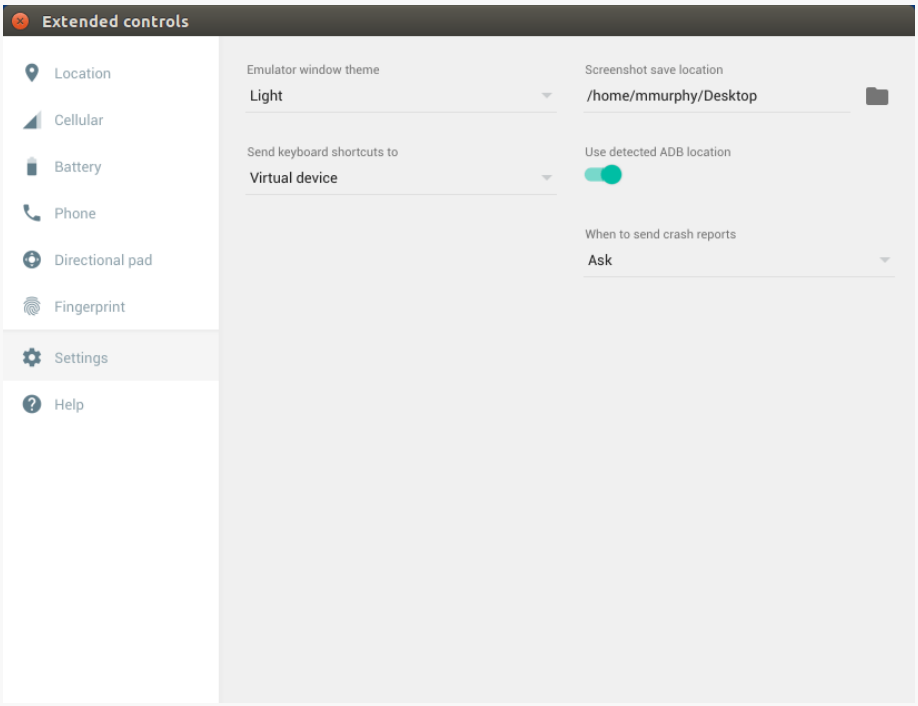


Figure 951: Emulator Extended Controls, Showing Settings

Faking the Real World

The “Extended controls” panel also allows you to fake real world behavior in your emulator. These are reminiscent of similar capabilities in the Android Device Monitor.

Location

The “Location” category lets you fake GPS fixes:

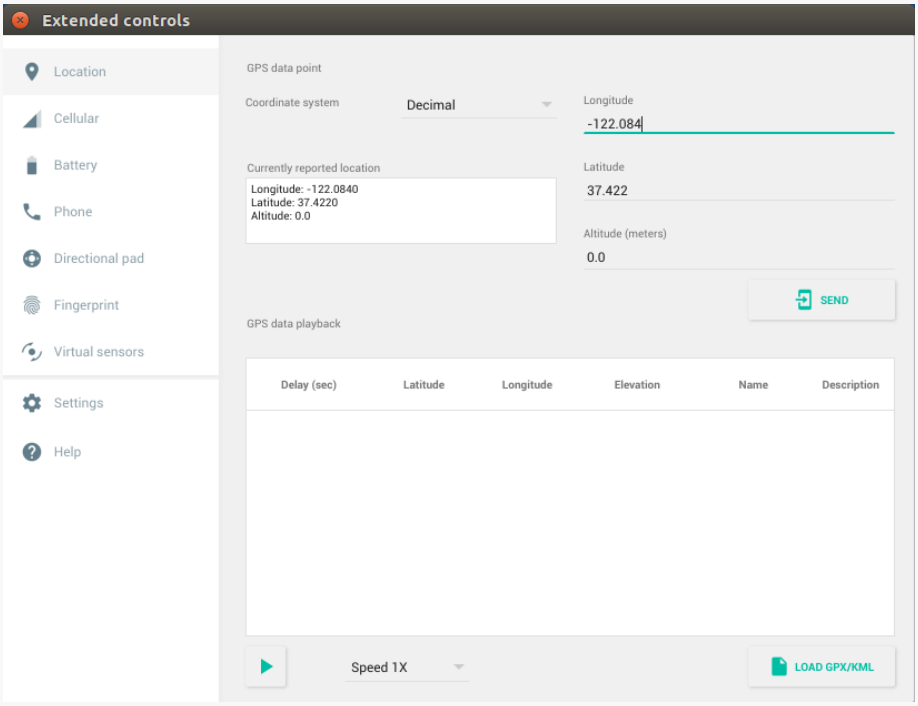


Figure 952: Emulator Extended Controls, Showing Location

The upper half allows you to specify a single GPS fix and “send” that to the emulator, which should respond the same way as if an actual Android device received a GPS fix.

The bottom half allows you to load a GPX or KML file containing a series of waypoints and the time between them, then play those back, either at normal speed or at an accelerated pace (if you get bored easily).

Network Status

The “Cellular” category controls how the emulator emulates its cellular network connection:

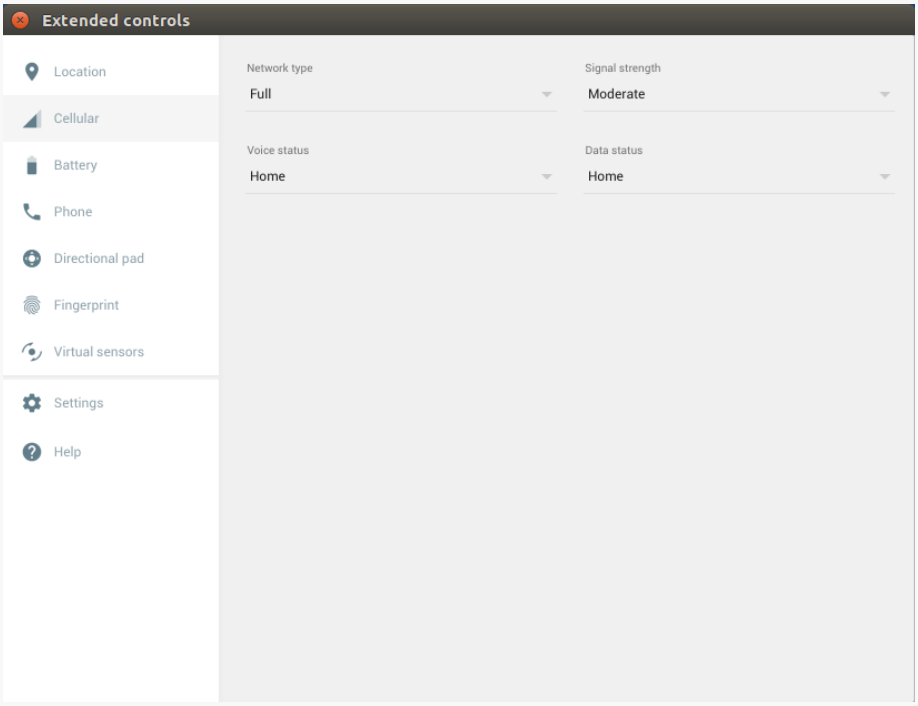


Figure 953: Emulator Extended Controls, Showing Cellular

Battery

The “Battery” category allows you to simulate changes in the power status of the emulator:

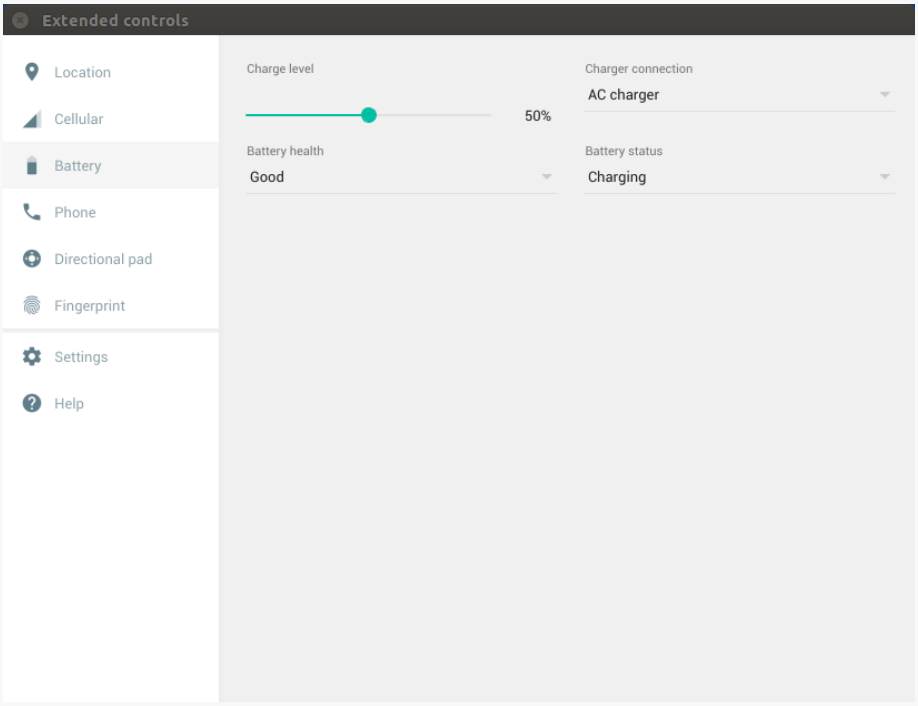


Figure 954: Emulator Extended Controls, Showing Battery

Not only will your code be able to receive events like `ACTION_BATTERY_CHANGED`, but you can see the changes in the status bar of the emulator, such as the battery icon showing charging status and current charge level.

Telephony

The "Telephony" category allows you to simulate incoming phone calls and text messages:

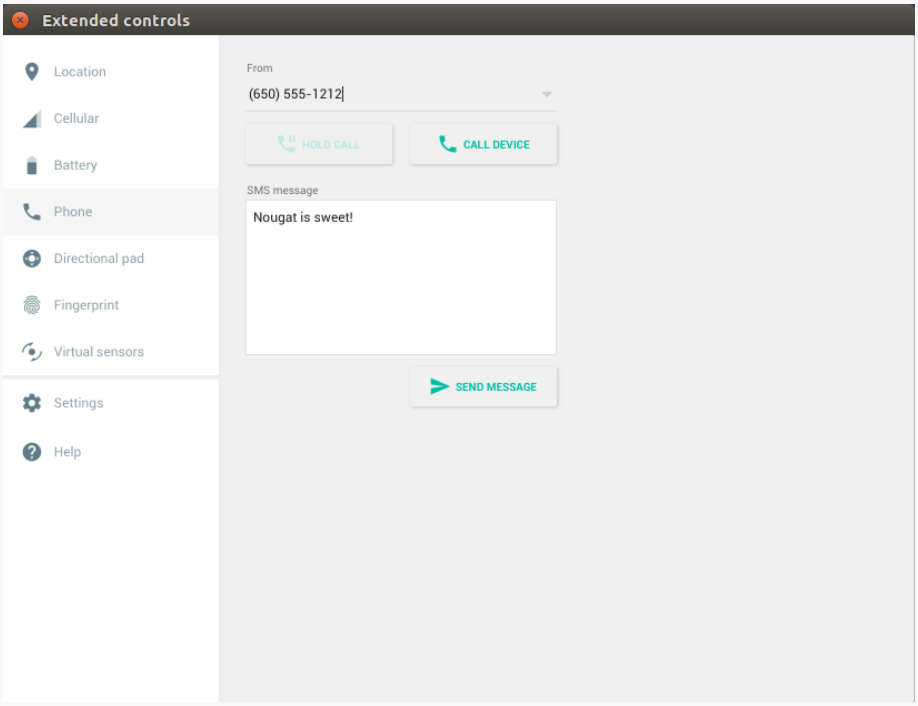


Figure 955: Emulator Extended Controls, Showing Telephony

Emulator Window Operations

Dragging a window edge of the emulator window will change the scale used by the emulator. The entire emulator window is still there, just smaller or larger than before. The resulting window will have the proper aspect ratio, so if you drag the left or right side and shrink the window, it will shrink both vertically and horizontally.

Using your development machine's native file manager (e.g., Nautilus on Ubuntu Linux), you can drag-and-drop files into the emulator window. If the file is an APK, it will be installed automatically, as if you had installed it through the `adb install` command. If the file is anything else, it will be uploaded into the emulator's `Download/` directory on external storage. If your app has permission to work with external storage, it can read the file from there.

Headless Operation

Sometimes, you want an emulator without a GUI. Typically, this is used for continuous integration or some other server-based testing solution — you use the "headless" emulator to run tests, even on a machine that lacks any GUI capability.

To do this, you will need to run the emulator from the command-line. Run `emulator -no-window -avd ...`, where `...` is the name of your AVD (e.g., the value in the left column of the list of AVDs in the AVD Manager). To test this first in normal mode, run the command without the `-no-window` switch.

The simplest solution to get rid of the emulator instance is to `kill` its process.

There are many other command-line switches for the emulator that you may wish to investigate. While most of these have UI analogues in the AVD Manager, the switches would be necessary to replicate some of those for headless operation.

Lint and the Support Annotations

As C/C++ developers are well aware, `lint` is not merely something that collects in pockets and belly buttons.

`lint` is a long-standing C/C++ utility that points out issues in a code base that are not errors or warnings, but are still indicative of a likely flaw in the code. After all, what might be legal from a syntax standpoint may still be a bug when used.

Android Studio and the Android Plugin for Gradle have their own equivalent Lint tool, for reporting similar sorts of issues with an Android project's Java code, resources, and manifest. You can also get Lint reports from the command line, such as via the Android Gradle Plugin, perhaps as part of integrating your builds into a continuous integration server.

To help Lint catch problems stemming from your own code, Google has released the `support-annotations` library, to help catch things like passing a widget ID, instead of a layout ID, into `setContentView()`. You can also use these annotations to help those using your code – whether in the same project or in consumers of a library that you publish – make sure that they do not make similar mistakes.

This chapter will explore how you use Lint to detect problems and how you can add annotations to your code to help Lint catch even more problems.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

What It Is

Lint can be best described as "a pest, but a good pest".

Normally, what stops you from building your app are compiler errors: bad Java syntax, malformed XML resource files, and the like. At the command line, these stop an in-progress build and dump error messages to the console. In Android Studio, these are noted in a log and also by notations in the source code, frequently as red sqiggle lines underneath the offending Java or XML when viewed in an editor. You also may get yellow squiggle lines for warnings — things the compiler will allow but the compiler thinks may be a problem.

However, there are many things that might be syntactically valid but are not a good idea from an Android standpoint. For example, if you specify a minimum SDK version of API Level 8, and you try using a class that only exists on API Level 11, that's a problem if you are not handling it correctly and avoiding this class on the older-yet-supported devices. Yet, if your build target (i.e., `compileSdkVersion` in Android Studio) is API Level 11 or higher, it is perfectly valid syntax and would compile just fine.

Lint is designed to encapsulate rules that transcend syntax, to add more errors and warnings that reflect good Android practices beyond simple validity.

When It Runs

Running Lint sometimes happens automatically (e.g., from your IDE) or sometimes happens manually. The following sections outline the various possibilities.

Android Studio

By default, in Android Studio, Lint will run when you save a file, giving you error (red) or warning (yellow) squiggles for things that run afoul of Lint rules:

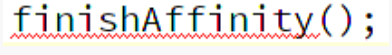


Figure 956: Android Studio Lint Error

You can manually invoke it via Analyze > Inspect Code... from the main menu, though this also performs other analyses that are not necessarily relevant for you as an Android developer, such as "J2ME issues".

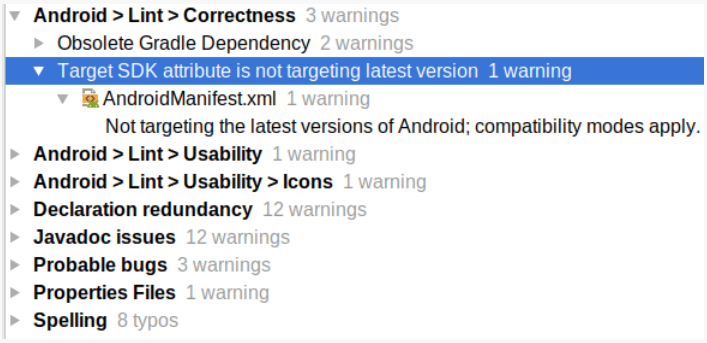


Figure 957: Android Studio Inspection Results

Command Line

You can also invoke Lint via `gradle lint` or a per-variant edition (e.g., `gradle lintRelease`). This will write results to an XML file in `build/outputs/` based upon product variant (e.g., `build/outputs/lint-results-release.xml` for a `gradle lintRelease` run). It will also emit an HTML file with the same base name in the same directory. These contain the same basic information as you get from the command-line output, with the XML in particular designed to be consumed by other tools, such as a continuous integration server.

What to Fix

In Android Studio, clicking on a red or yellow squiggle will pop up an adjacent "lightbulb" drop-down offering ways to fix the problem:

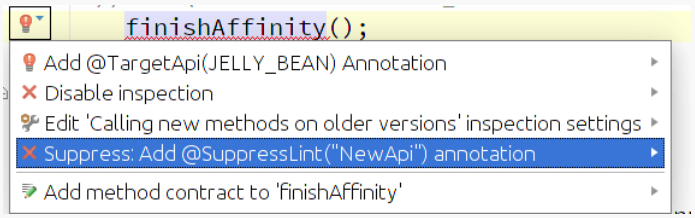


Figure 958: Android Studio Lint Fix Suggestions

You can also bring up this "quick fixes" list via Alt-Enter. For example:

- Errors related to accessing classes or methods higher than your `minSdkVersion` have "quick fixes" to add the `@TargetApi` annotation to the class or method containing your code

- Warnings related to hard-coded strings in layouts or the manifest have “quick fixes” to convert those strings into string resources

All warnings and errors will have “quick fixes” to suppress that warning or error in the future, by adding notations to the file to that effect.

What to Configure

You have some measure of control over Lint’s behavior, though the mechanics of doing this varies by tool.

Android Studio

In Android Studio, you can configure Lint’s behavior via the Project Settings dialog, accessible via File > Settings:

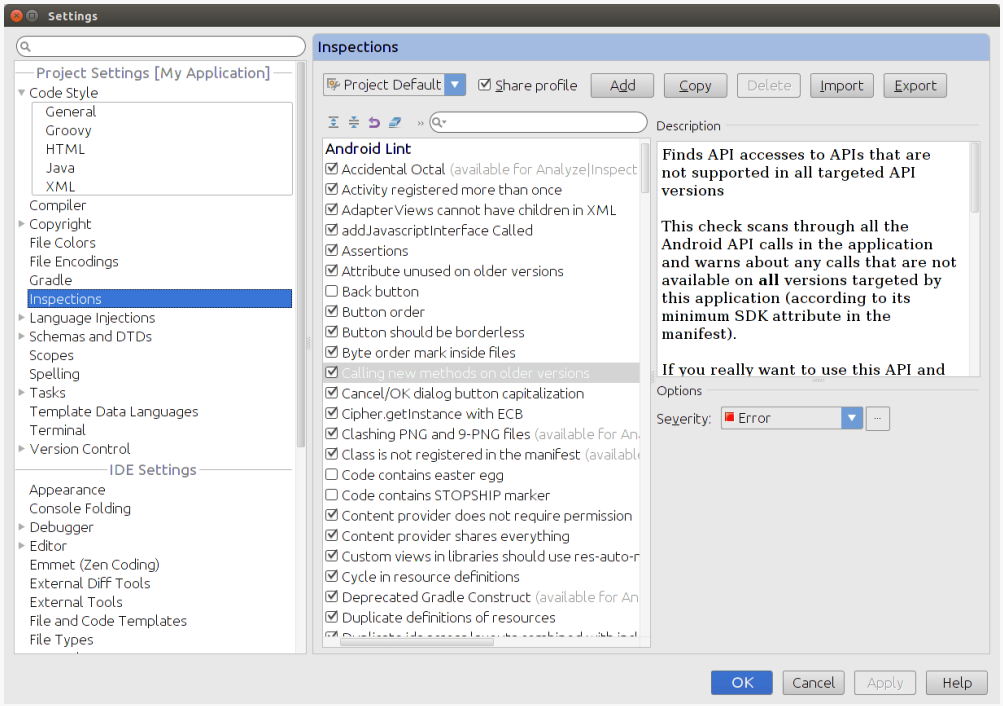


Figure 959: Android Studio Lint Error Checking Preferences

You can change some details about the specific checks that Lint makes:

- the severity of the issue, usually set to Warning or Error

- whether the specific issue should be ignored rather than executed

To do this, you may wish to create your own inspection profile, rather than modifying the stock “Project Default” profile. To do this, just click the “Copy” button in the Inspections page of the Settings dialog and supply a name for the new profile.

The above recipe changes the inspections for the individual project. To change them for new projects, go into File > Other Settings > Default Settings, and make your changes there.

Command Line

To block certain Lint checks in Gradle, you can create a `lint.xml` file, in the root directory of your project, containing information about which particular issues should be suppressed for that project. The benefit here is that you can configure suppression at a finer granularity, blocking issues for certain files or directories and allowing them for others. The sample `lint.xml` from [the Lint documentation] looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<lint>
    <!-- Disable the given check in this project -->
    <issue id="IconMissingDensityFolder" severity="ignore" />

    <!-- Ignore the ObsoleteLayoutParam issue in the given files -->
    <issue id="ObsoleteLayoutParam">
        <ignore path="res/layout/activation.xml" />
        <ignore path="res/layout-xlarge/activation.xml" />
    </issue>

    <!-- Ignore the UselessLeaf issue in the given file -->
    <issue id="UselessLeaf">
        <ignore path="res/layout/main.xml" />
    </issue>

    <!-- Change the severity of hardcoded strings to "error" -->
    <issue id="HardcodedText" severity="error" />
</lint>
```

You can also configure lint via a `lintOptions` closure inside the `android` closure of your `build.gradle` file. In particular, you can have a `disable` statement to list the Lint checks that you would like to block:

```
android {
    lintOptions {
        disable 'IconMissingDensityFolder','InefficientWeight'
        ...
    }
}
```

The names used in `lint.xml` or `lintOptions` are the "issue IDs". You can get a roster of these by running `lint --list` for brief summaries:

```
Valid issue categories:
    Correctness
    Correctness:Messages
    Security
    Performance
    Usability:Typography
    Usability:Icons
    Usability
    Accessibility
    Internationalization
    Bi-directional Text

Valid issue id's:
"ContentDescription": Image without contentDescription
"AddJavascriptInterface": addJavascriptInterface Called
"ShortAlarm": Short or Frequent Alarm
"AlwaysShowAction": Usage of showAsAction=always
"ShiftFlags": Dangerous Flag Constant Declaration
"LocalSuppress": @SuppressLint on invalid element
"UniqueConstants": Overlapping Enumeration Constants
"InlinedApi": Using inlined constants on older versions
"Override": Method conflicts with new inherited method
"NewApi": Calling new methods on older versions
...
```

...or `lint --show` for a set of more elaborate descriptions:

```
Available issues:

Correctness
===========

AdapterViewChildren
-------------------
Summary: AdapterViews cannot have children in XML

Priority: 10 / 10
```

```
Severity: Warning
Category: Correctness

AdapterViews such as ListViews must be configured with data from Java code,
such as a ListAdapter.

More information:
http://developer.android.com/reference/android/widget/AdapterView.html

OnClick
-------
Summary: onClick method does not exist

Priority: 10 / 10
Severity: Error
Category: Correctness

The onClick attribute value should be the name of a method in this View's
context to invoke when the view is clicked. This name must correspond to a
public method that takes exactly one parameter of type View.

Must be a string value, using '\;' to escape characters such as '\n' or
'\uxxxx' for a unicode character.


StopShip
--------
Summary: Code contains STOPSHIP marker

Priority: 10 / 10
Severity: Warning
Category: Correctness
NOTE: This issue is disabled by default!
You can enable it by adding --enable StopShip

Using the comment // STOPSHIP can be used to flag code that is incomplete but
checked in. This comment marker can be used to indicate that the code should
not be shipped until the issue is addressed, and lint will look for these.


MissingPermission
-----------------
Summary: Missing Permissions

Priority: 9 / 10
Severity: Error
Category: Correctness
```

```
This check scans through your code and libraries and looks at the APIs being
used, and checks this against the set of permissions required to access those
APIs. If the code using those APIs is called at runtime, then the program will
crash.
...
```

The `lint` command can be found in the `tools/` directory of your Android SDK installation.

Support Annotations

The `support-annotations` library, from the Android Support set of libraries, offers a series of annotations that you can add to methods, method parameters, and the like to teach Lint certain types of bugs to check for. Some of the Android Support libraries use these annotations, so Lint can help catch problems when you use those public APIs. You, in turn, can add these annotations to *your* code, to catch certain problems at compile time that otherwise might be missed.

However, the important thing is that these are compile-time checks, not assertions at runtime. Lint will see if there is a likely bug at compile time and point it out to the developer, but there are many places where Lint simply has no way to know if everything is OK or not. These annotations are not a replacement for defensive programming. In fact, they not only help *users* of some API you publish to use it correctly, they help *you* by serving as a reminder that these should be checked at runtime as well.

Pretty much all of the Android Support libraries pull in `support-annotations`, courtesy of Gradle and transitive dependencies. If you do not seem to have `support-annotations` in your project, just add it to your `dependencies` closure, as you would any other of the Android Support libraries:

```
dependencies {
  implementation 'com.android.support:support-annotations:23.1.0'
}
```

You may occasionally run into version conflicts over this library, where Library A wants one version and Library B wants another version. In those cases, you may need to teach Gradle to not try to load the `support-annotations` version that a particular library might want, so you can use a different version:

```
dependencies {
  implementation 'com.android.support:support-annotations:23.1.0'
```

```
  compile('com.davemorrissey.labs:subsampling-scale-image-view:3.4.0') {
    exclude module: 'support-annotations'
  }
}
```

In this case, `com.davemorrissey.labs:subsampling-scale-image-view:3.4.0` wants version `20.0.0` of the `support-annotations`, which is rather old. Hence, we block that dependency and substitute our own, for version `23.1.0`. In general, newer versions of this library should be backwards-compatible with older versions of this library, so in case of conflict, use the newer version.

Permissions, Again

You can indicate that certain bits of your app require callers to hold certain permissions, using the `@RequiresPermission` annotation. This is mostly for libraries, where other projects might use the library.

Methods

The most common place to put this annotation will be on a method, to indicate that the method requires that callers hold a certain permission.

The simplest scenario is where the method requires that callers hold a single permission, in which case you just list the permission as a parameter to the annotation:

```
@RequiresPermission(Manifest.permission.CAMERA)
public void takeSelfie() {
  // do work here
}
```

If the caller does not have a `<uses-permission>` element for the `CAMERA` permission, Lint will complain at the point where the app calls `takeSelfie()`.

Sometimes, you may need callers to hold more than one permission. In that case, you can use `allOf` to list permissions; callers have to have requested all of them in the manifest:

```
@RequiresPermission(
  allOf = {
    Manifest.permission.CAMERA,
    Manifest.permission.WRITE_EXTERNAL_STORAGE
  }
```

```
)
public void takeSelfie() {
  // do work here
}
```

On occasion, you may need the caller to hold one of a set of possible permissions. The quintessential example here is location, where your code might dynamically adapt based upon whether the app has `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. In that case, you can use `anyOf` to list the possibilities; if the app has any of those permissions requested in the manifest, Lint will be happy:

```
@RequiresPermission(
  anyOf = {
    Manifest.permission.ACCESS_COARSE_LOCATION,
    Manifest.permission.ACCESS_FINE_LOCATION
  }
)
public void makeNoteOfWhereWeAt() {
  // do work here
}
```

It is unclear if there is a way to combine `anyOf` and `allOf` in a single annotation (e.g., a `takeSelfie()` method that wants to geotag the photo with the user's current location).

Intent Actions

If you have a custom `Intent` action string, and the operation tied to that action string requires a permission, you can teach Lint about that by putting a `@RequiresPermission` annotation on a `static String` for that action string:

```
@RequiresPermission(Manifest.permission.CAMERA)
public static final String ACTION_TAKE_SELFIE="com.commonsware.intent.action.SELFIE";
```

Basically, Lint keeps track of such strings, and if any `Intent` in the app is created using those strings as actions, Lint will check to see if the permission was requested.

ContentProviders

Similarly, you can annotate a `static Uri` that serves as the base `Uri` for a `ContentProvider`. Any calls to `ContentObserver` that have a `Uri` based on the `static` base will trigger Lint to check to see if permissions were requested.

Frequently, though, a `ContentProvider` will have separate read and write permissions. To handle that, you have to use some fairly clunky syntax, wrapping the `@RequiresPermission` annotation in `@RequiresPermission.Read` or `@RequiresPermission.Write` annotations. For example, the `ContactsContract` class could, in theory, have:

```
public static final String AUTHORITY = "com.android.contacts";
public static final Uri AUTHORITY_URI = Uri.parse("content://" + AUTHORITY);

@RequiresPermission.Read(@RequiresPermission(Manifest.permission.READ_CONTACTS))
@RequiresPermission.Write(@RequiresPermission(Manifest.permission.WRITE_CONTACTS))
public static final Uri CONTENT_URI = Uri.withAppendedPath(AUTHORITY_URI, "contacts");
```

(it actually does not have these; any Lint checks for this `CONTENT_URI` are being handled through rules internal to the tools, not through the support annotations)

What Permissions Should I Annotate?

If the method (or whatever) absolutely needs the permission, in all significant cases, then having the annotation will be useful.

However, there will be scenarios in which a permission may or may not be needed, depending upon circumstances.

For example, let's go back to:

```
@RequiresPermission(
  allOf = {
    Manifest.permission.CAMERA,
    Manifest.permission.WRITE_EXTERNAL_STORAGE
  }
)
public void takeSelfie() {
  // do work here
}
```

Here, we are implying that `takeSelfie()` will need both of those permissions, and probably all of the time. For example, perhaps the method is set up to write to `Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DCIM)`. That directory requires `WRITE_EXTERNAL_STORAGE` all of the time.

But, suppose the method were implemented where the destination was a `Uri`, instead? You would have:

```
@RequiresPermission(
  allOf = {
    Manifest.permission.CAMERA,
    Manifest.permission.WRITE_EXTERNAL_STORAGE
  }
)
public void takeSelfie(Uri dest) {
  // do work here
}
```

That `Uri` could point to any number of locations, only some of which might require `WRITE_EXTERNAL_STORAGE`. For example, the caller could provide a `file: Uri` pointing to `getExternalFilesDir()`, which does not need `WRITE_EXTERNAL_STORAGE` on API Level 19+.

There are two major strategies here, with respect to these annotations:

- Be conservative, and annotate for both permissions, as shown in the example above. The caller can always suppress the warning, if the caller is sure that `WRITE_EXTERNAL_STORAGE` is not required. However, this may confuse people not familiar with your API or with Android overall.
- Be liberal, and only annotate for the `CAMERA` permission (which `takeSelfies()` always needs). Here, you are relying on the caller to read the documentation for your library, use common sense, or perform adequate testing to ensure that `WRITE_EXTERNAL_STORAGE` is requested in cases where it is needed.

A "middle ground" approach would be to be conservative in cases where the permission might require significant work, and liberal otherwise. For example, `WRITE_EXTERNAL_STORAGE` is a `dangerous` permission on Android 6.0+, and so the caller has to go through all of the runtime permission request stuff for that if the app's `targetSdkVersion` is 23 or higher. But, if `takeSelfie()` really needed `CAMERA` and `VIBRATE` (to shake the device once the selfie is taken, perhaps based on user preferences), requesting `VIBRATE` is merely a single line in the manifest, and so demanding it via the annotation when it might not be needed would be excessive.

Type Roles, and the War on Enums

In 2015, a kerfuffle erupted in the world of Android development, one that quickly got tagged with the label, "the War on Enums".

Google developer advocates started promoting the idea that using the Java `enum` construct was bad, and that you should use `int` constants instead, the way the Android SDK does. Core Android engineers slowly backed away from those developer advocates, but explained the reason why all through the Android SDK we are passing around `int` values.

In a nutshell, an `enum` reference will consume more heap space than will an `int`. If every place we passed around `int` flags or `int` resource IDs, we passed around `enum` objects, we would put greater pressure on our available heap space.

For most Android developers, for their own code, this particular concern is unimportant, compared to the type safety one gets from using an `enum` properly. However, the Android SDK team decided that, in general, they should use `int` values rather than `enum` values, so they would not be the ones to blame for consuming too much heap space.

However, this does bring us back to the core problem of passing the *wrong* `int` values into the *wrong* methods, such as:

- passing a widget ID or a string resource ID into `setContentView()`
- passing in `Intent` flags (e.g., `FLAG_ACTIVITY_NEW_TASK`) to `PendingIntent` methods like `getActivity()`
- passing in a color resource ID to a method that takes an actual ARGB color value

Instead, we get a convoluted set of annotations to try to help developers using public APIs to provide the smarts that ordinarily would be handled simply by `enum`.

Resources

Ideally, resource IDs would use Java's `enum`, so that you could not pass a string resource ID to a method that is expecting a menu resource ID. Alas, that is not the case.

Instead, if you accept resource IDs as parameters on methods or as return values from those methods, you can use a set of annotations to indicate what specific role the `int` values play.

- `@AnimatorRes`
- `@AnimRes`
- `@ArrayRes`

- `@AttrRes`
- `@BoolRes`
- `@ColorRes` (i.e., the `int` should be a color resource ID)
- `@DimenRes`
- `@DrawableRes` (i.e., the `int` should be a drawable resource ID)
- `@FractionRes`
- `@IdRes`
- `@InterpolatorRes`
- `@LayoutRes`
- `@MenuRes`
- `@PluralsRes`
- `@RawRes`
- `@StringRes` (i.e., the `int` should be a string resource ID)
- `@StyleableRes`
- `@StyleRes`
- `@TransitionRes`
- `@XmlRes`

Documentation for these, such as it is, can be found in the JavaDocs for the `android.support.annotation` package.

There is also `@AnyRes`, which indicates that the `int` needs to be a resource, but does not imply a particular type of resource.

So, for example, you could have:

```
public void loadConfig(@XmlRes int xmlResId) {
  // do work here
}
```

If Lint is uncertain whether the parameter passed to `loadConfig()` is really an `R.xml` value, it can warn the caller.

Custom Enum Replacement

Sometimes, the `int` values are replacing what would have been a custom Java `enum`.

For example, the CWAC-Cam2 library has a `FlashMode` `enum`:

```
public enum FlashMode {
  OFF,
  ALWAYS,
```

```
  AUTO,
  REDEYE
}
```

Apparently, the author of that library is evil and therefore supports the use of an `enum`.

(note: the author of that library is also the author of this book)

An alternative would be to define those as a series of `int` flags:

```
public class FlashMode {
  public static final int OFF=0x0;
  public static final int ALWAYS=0x1;
  public static final int AUTO=0x2;
  public static final int REDEYE=0x3;
}
```

However, we are then back in the state where we do not know if some arbitrary `int` that we are passed as a parameter really is a `FlashMode`, as a method might expect. With the `enum`, we can have methods like:

```
public void setFlashMode(FlashMode mode) {
  // do work
}
```

The `support-annotations` library makes it possible to write a `setFlashMode()` that warns developers if they pass in the wrong `int`, but it takes a bit of work.

The documented recipe is:

```
public class FlashMode {
  @IntDef({OFF, ALWAYS, AUTO, REDEYE})
  @Retention(RetentionPolicy.SOURCE)
  public @interface FlashModeInt {}

  public static final int OFF=0x0;
  public static final int ALWAYS=0x1;
  public static final int AUTO=0x2;
  public static final int REDEYE=0x3;
}
```

Then, elsewhere, we could reference that custom `FlashModeInt` annotation:

```
public void setFlashMode(@FlashMode.FlashModeInt int mode) {
  // do work
}
```

Then, if Lint cannot confirm that the supplied `mode` is one of those constants, Lint can warn the caller.

Flags

One benefit of `int` over `enum` is that it is easier to implement parameters and return values that represent a combination of values rather than single values.

For example, there are a wide range of flags that you can put on an `Intent`, like `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_SINGLE_TOP`. An `Intent` can have zero, one, or several of these flags.

With an `enum` for those flags, you would need to be passing around a `Set` of `enum` instances. With `int` values, though, you can use bitfields, where each flag is assigned a bit within the `int`. For example, `FLAG_ACTIVITY_CLEAR_TOP` is `0x04000000` and `FLAG_ACTIVITY_SINGLE_TOP` is `0x20000000`. Having both of those on a single `Intent` is merely a matter of using a OR bit operation:

```
yourIntent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP|FLAG_ACTIVITY_SINGLE_TOP)
```

This takes up a lot less space, and is more efficient from a CPU standpoint, than a `Set` of `enum` values. However, once again, type safety becomes a problem.

`@IntDef` also supports a "flag" mode, where Lint will validate that the value passed in is comprised of the designated constants, either used individually or in combination using boolean bit operators. For example, perhaps we can support several possible flash modes in a camera API, and the caller can indicate the various modes of interest using flags:

```
@IntDef(flag=true, value={
  FLAG_OFF,
  FLAG_ALWAYS,
  FLAG_ON,
  FLAG_REDEYE
})
@Retention(RetentionPolicy.SOURCE)
public @interface FlashModeOptions {}
```

Now, methods and parameters annotated with `@FlashModeOptions` will be validated to ensure they are passing valid flags and combinations of flags.

Does It Null?

The `@NonNull` annotation can be used for parameters that are not allowed to be `null`. If, at compile time, the caller is clearly passing a `null` value, the caller will be warned.

```
public void doSomethingContextual(@NonNull Context ctxt) {
  // do work here
}
```

This could also be used on methods where you are sure that the return value cannot be `null`. This is particularly important with abstract methods, methods you expect other developers to override, callbacks, and the like — putting `@NotNull` on return values for those methods indicate that *you* are requiring that the implementer not hand you back a `null` value.

Conversely, the `@Nullable` annotation can be used on methods that explicitly *can* return `null` as valid value:

```
@Nullable
public Context getContextIfWeHaveOne() {
  // do work here

  return(result);
}
```

Any caller of `getContextIfWeHaveOne()` will get a Lint warning, pointing out that they need to check for `null` results. That warning will remain there until the developer suppresses it or, in Lint's estimation, appears to check the result for a `null` value and handle that case.

This can be used to help find `@NonNull` violations elsewhere, by helping Lint see where things might be `null`.

Data Validation

A variety of other annotations can be used for checking parameter values at compile time, to perhaps catch bugs earlier.

Size

For parameters and return values that implement `java.util.Collection` – such as `ArrayList`, you can use the `@Size` annotation to provide some compile-time guidance with regards to your expectations for that collection. This also works for ordinary Java arrays.

A simple number in the `@Size` annotation means you are expecting exactly that number of items in the collection, no more, no less:

```
public void growPair(@Size(2) ArrayList<String> values) {
  // do something
}
```

You can use `min` and `max` to constrain the size, without tying it down to a particular value:

```
public void sortInPlace(@Size(min=1) List<Comparable> unsorted) {
  // do a sort
}

public @Size(min=1, max=6) float[] getReading(SensorEvent e) {
  return(e.values);
}
```

Occasionally, you might have a collection that does not have a specific size, but the size it *does* have has to be evenly divisible by some number. For that, there is the `multiple` option:

```
@RequiresPermission(Manifest.permission.VIBRATE)
public void shakeItOff(@Size(multiple=2) long[] vibrationPattern) {
  // use Vibrator system service
}
```

Ranges

`@IntRange` and `@FloatRange` help validate that the annotated value lies within a particular range of values. The range is inclusive: values equal to the ends of the range are assumed to be valid.

These work somewhat like `@Size`, except they directly examine a value, instead of examining the length of a collection or string.

```
public void howManyRoadsMustAManWalkDown(@IntRange(from=0,to=42) int roads) {
  // do something involving a towel
}
```

Colors

If you have a method that expects a color resource ID as a parameter or return value, use the `@ColorRes` annotation, as noted previously.

However, more often than not, you will be expecting *colors*, not color resource IDs, to give the other developers flexibility about where the colors come from. In that case, `@ColorInt` will help identify parameters and return values that are expected to be actual ARGB colors, not just arbitrary integers. In particular, this will catch when somebody tries using a color resource ID where you expect an actual color.

Thread Validation

If a method needs to be invoked on a certain type of thread (e.g., a background thread), you can use annotations to try to catch that sort of bug.

The simple one is `@WorkerThread`, which indicates that the method needs to be called on a background thread. If Lint thinks that the method is being invoked from something else (e.g., the main application thread), it will flag the caller with a warning.

```
@WorkerThread
public void thisIsGoingToTakeLikeForEVER() {
  // do something tedious
}
```

There are two possible converse annotations: `@MainThread` and `@UiThread`. In one bit of documentation, Google says they are interchangeable. In another bit of documentation, Google tries to point out a disparity between them

> There is one and only one main thread in the process. That's the @MainThread. That thread is also a @UiThread. This thread is what the main window of an activity runs on, for example. However it is also possible for applications to create other threads on which they run different windows. This will be very rare; really the main place this distinction matters is the system process. Generally you'll want to annotate methods associated with the life cycle with @MainThread, and methods associated with the view hierarchy with @UiThread. Since the @MainThread is a

> @UiThread, and since it's usually the case that a @UiThread is the @MainThread, the tools (lint, Android Studio, etc) treat these threads as interchangeable, so you can call @UiThread methods from @MainThread methods and vice versa.

Roughly speaking, if the method has to be run on the main application thread for lifecycle reasons, use `@MainThread`. If the method has to be run on a UI thread to avoid "cannot modify views from a non-UI thread" sorts of errors, use `@UiThread`. And, if you're not sure, flip a coin.

Other Annotations

If you have a `protected` or `public` method in a class that might be subclassed, and you want to help ensure that if the method is overridden that the developer calls through to your superclass implementation, use `@CallSuper`.

(note: this annotation will not call a building superintendent; it will only be honored by Superman if your name is on the whitelist, e.g., Lois Lane)

```
@CallSuper
protected int heyDontForgetAboutMe() {
  // do something

  return(somethingToo);
}
```

`@CheckResult` allows you to nag any caller of your method, to ensure that they actually look at the value you return, rather than ignore it.

Inspecting Layouts

Layouts get complicated. Not only might you be pulling in from several sources (via fragments or `<include>` or whatever), but you make changes to the contents of the UI at runtime. What you have in your layout resources is a starting point, but only that. Sometimes, it would be helpful to see exactly what is in the UI of your app *right now*, based on what you have done inside that app.

Android Studio, starting with 2.2, has a built-in layout inspector that will help with this process. Beyond that, Android comes with a Hierarchy View tool, designed to help you visualize your layouts as they are seen in a running activity in a running emulator or Android 4.1+ device. So, for example, you can determine how much space a certain widget is taking up, or try to find where a widget is hiding that does not appear on the screen.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

The Layout Inspector

Users of Android Studio, starting with version 2.2, have access to a Layout Inspector that captures the state of the foreground activity and all of its widgets. You can examine those widgets to try to determine why your UI perhaps is not working in the way that you expect.

Launching the Layout Inspector

First, get a debuggable build of your app running on your chosen device or emulator, with the desired activity in the foreground, in the state that you are interested in. Layout Inspector captures a snapshot of the state; it does not continuously update as the UI changes. Hence, you need to get the UI into the state that you want to inspect first.

Then, choose Tools > Android > Layout Inspector from the Android Studio main menu.

Viewing the View Hierarchy

When you inspect a layout, a tab is opened in Android Studio with a tree of widgets on the left, a wireframe-enhanced screenshot in the center, and a properties pane on the right:

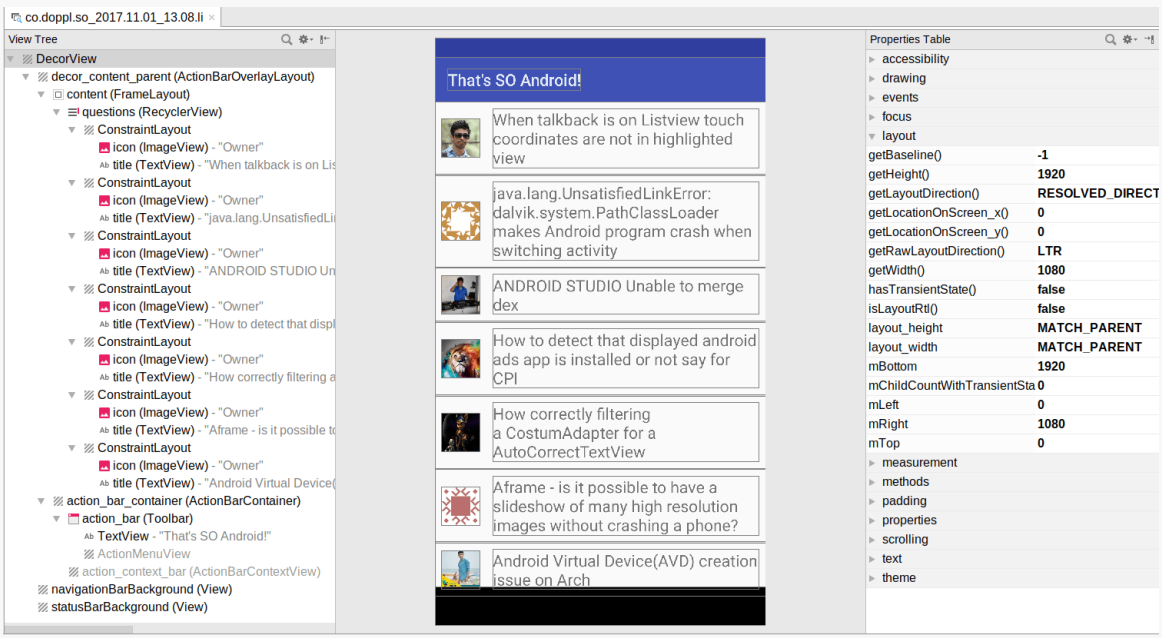


Figure 960: Layout Inspector, As Initially Launched

Hovering over a widget in the tree or the wireframe will outline it in red. Selecting that widget in the tree or the wireframe, by clicking on it, outlines it in blue and updates the properties pane with the properties for that particular widget or container.

The properties pane shows a tree of fields, getter methods, and XML-style attributes. You can use these to view the state of the widget or container. However, these properties are read-only; you cannot modify the UI from the Layout Inspector.

Inspections and Captures

Layout Inspector is not purely a tab in the IDE. The data collected by Android Studio to populate the Layout Inspector is saved in your project's `captures/` directory. This allows you to view that same data again in the future. This may be useful for comparing different captures from different times, such as "before-and-after" captures to see what effects a code change had on your layout contents.

Hierarchy View

Hierarchy View is the original incarnation of this sort of layout inspection tool. Hierarchy View was part of the Android Developer Tools (ADT) plugin for Eclipse, back when Google supported Eclipse. Android Studio users may prefer Layout Inspector, but Hierarchy View is still available — for a while at least — for those seeking an alternative.

Where Hierarchy View Works

Hierarchy View originally only worked with the Android SDK emulator.

However, starting with Android 4.1, you can use Hierarchy View with devices. To make this work, you need to define an `ANDROID_HVPROTO` environment variable on your development machine, setting it equal to `ddm` (e.g., `export ANDROID_HVPROTO=ddm` on macOS or Linux).

If you need to test on Android 4.0 or older hardware, you need to use `ViewServer`, described later in this chapter.

Launching Hierarchy View

To launch Hierarchy View, you can use the Android Device Monitor. In Android Studio, this is available from the Tools > Android menu. From the command line, run the `monitor` program to bring up the Android Device Monitor. In the Android Device Monitor, choose Window > Open Perspective from the main menu, and open Hierarchy View:

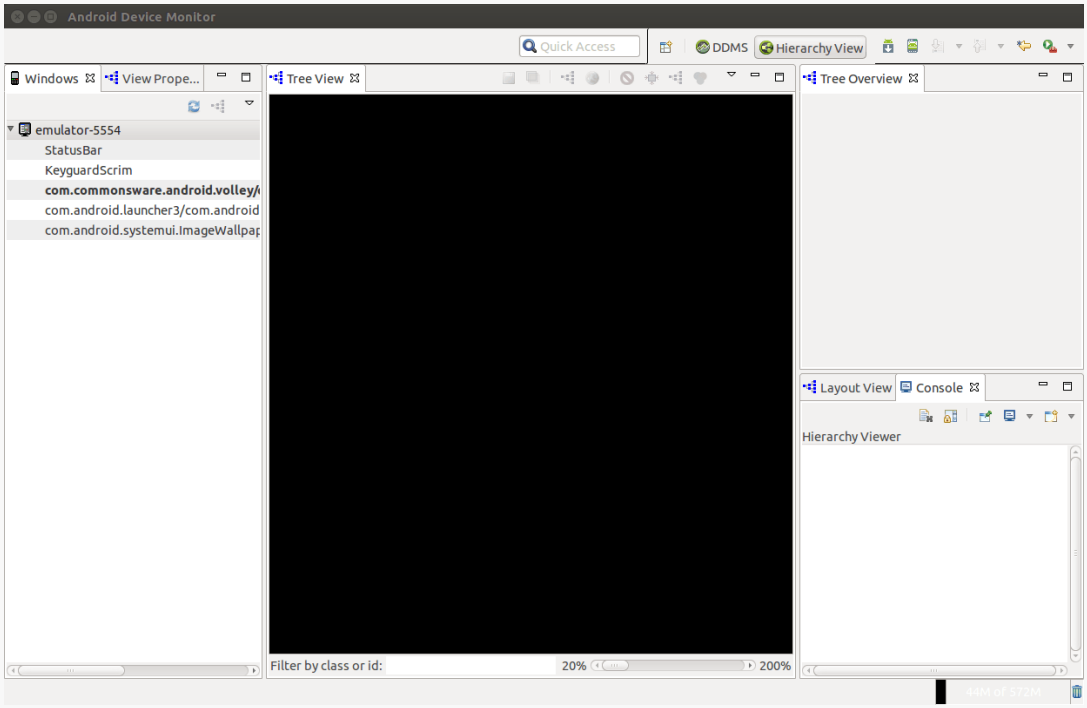


Figure 961: Hierarchy View As Originally Opened

The roots of the tree-table on the left show the emulator instances presently running on your development machine. The leaves represent applications running on that particular emulator. Your activity will be identified by application package and class (e.g., `com.commonsware.android.files/...`).

Viewing the View Hierarchy

Where things get interesting, though, is when you double-click on your activity in the tree-table. After a few seconds, the details spring into, er, view:

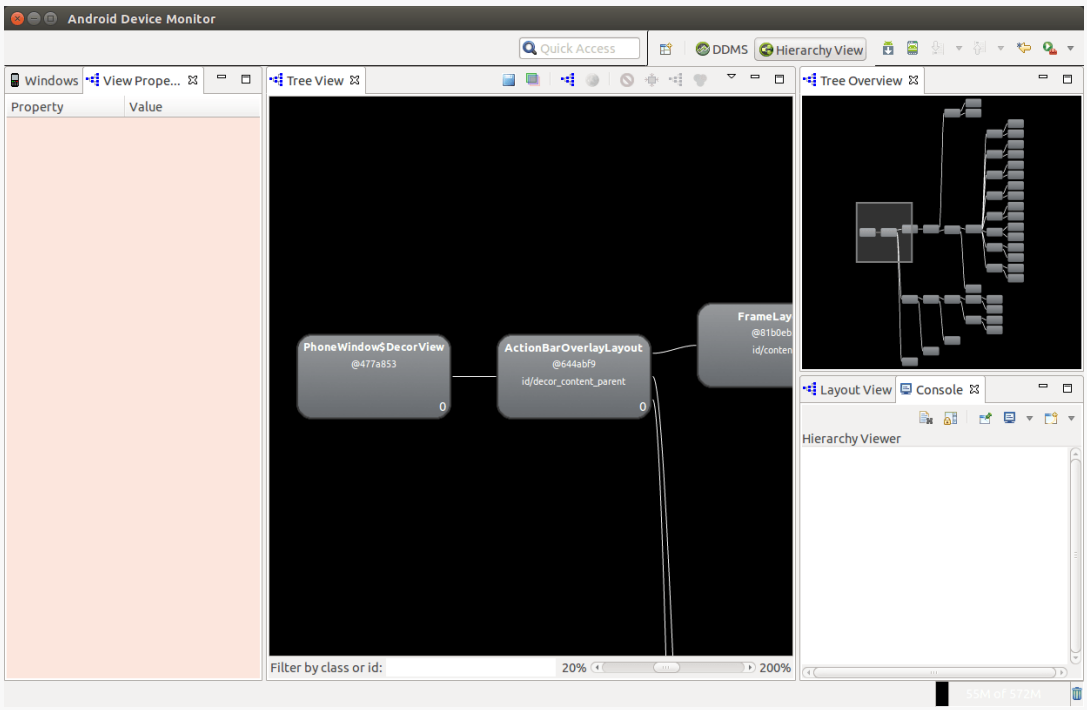


Figure 962: Hierarchy View Showing an Activity

The main area of the Layout View shows a tree of the various widgets and stuff that make up your activity, starting from the overall system window and driving down into the individual UI widgets that users are supposed to interact with. This includes both widgets and containers defined by your application and others that are supplied by the system, including the title bar.

Clicking on one of the views adds more information to this perspective:

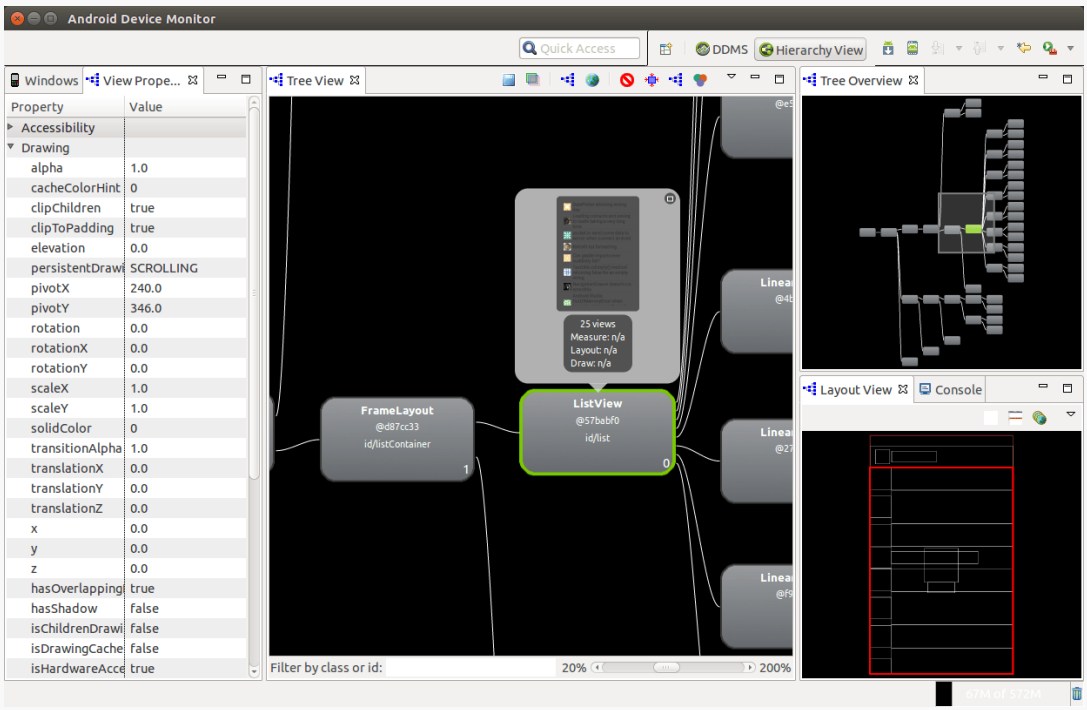


Figure 963: Hierarchy View Showing a View's Details

Now, we get:

- In the left region of the Viewer, we see the properties of the selected widget or container, in its own tree-table.
- In the Tree View in the middle, the selected widget or container has a pop-up bubble with what that particular `View` looks like on the screen, along with some performance timing data.
- In the Tree Overview in the upper-right portion of the tool, our selected `View` is highlighted in green.
- In the Layout View in the lower-right portion of the tool, our selected `View` is highlighted in red in the wireframe.

From the toolbar above the Tree View, you can:

- Save the tree diagram as a PNG file
- Save the UI as a Photoshop PSD file, with different layers for the different widgets and containers
- Force the UI to repaint in the emulator or re-load the hierarchy, in case you have made changes to a database or to the app's contents and need a fresh diagram

ViewServer

One original limitation of Hierarchy View was that it only worked with the emulator. There was no means for it to pull information from random activities running on production hardware. And while this restriction was lifted starting with Android 4.1, you may still need to use Hierarchy View on Android 4.0 or older hardware.

Romain Guy, one of the early Android engineers, published a `ViewServer` open-source component that gets around this limitation.

If you add the `ViewServer` source code to your project, and register your activities as they are created (and remove them when they are destroyed), you will be able to use Hierarchy View with them. However, this is a bit dangerous on a production app, so you should strongly consider using `BuildConfig.DEBUG` to only enable this logic in debug builds.

Blending in the `BuildConfig.DEBUG` concept with Mr. Guy's supplied sample usage, we get something like this:

```
public class MyActivity extends Activity {
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);

     // Set content view, etc.

     if (BuildConfig.DEBUG) ViewServer.get(this).addWindow(this);
   }

   public void onDestroy() {
     if (BuildConfig.DEBUG) ViewServer.get(this).removeWindow(this);

     super.onDestroy();
   }

   public void onResume() {
     super.onResume();

     if (BuildConfig.DEBUG) ViewServer.get(this).setFocusedWindow(this);
   }
}
```

Also note that `ViewServer` requires that your application hold the `INTERNET` permission, which you may already have requested for other reasons.

Screenshots and Screencasts

They say that a picture is worth a thousand words.

If that were really true, this book would be a lot shorter, mostly consisting of a bunch of screenshots.

That being said, having screenshots of your app is essential for documentation, marketing, and other uses. You are going to want to collect screenshots from your app by one means or another.

Screencasts — videos recording the user's interaction with a device – are also very useful for the same purposes, even if their nature precludes their practical use in various mediums (e.g., PDFs). These are also a bit more complex to collect, though you have plenty of options for that.

This chapter will outline various ways to get screenshots and screencasts of your app.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Collecting from Android Studio

The Logcat tool has buttons to take a screenshot and record a screencast, in the toolbar:

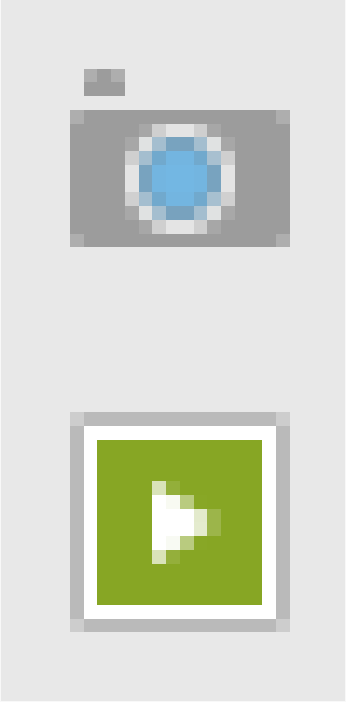


Figure 964: Android Studio Screenshot and Screencast Toolbar Buttons

Note that these toolbar buttons may be hidden, as they are fairly far down the toolbar. You may need to expand the Logcat pane to show more of the toolbar to get to these buttons. Or, there is a “>>” affordance that shows the hidden toolbar buttons that you can click.

Screenshots

The top one takes a screenshot, giving you a dialog to control what gets captured:

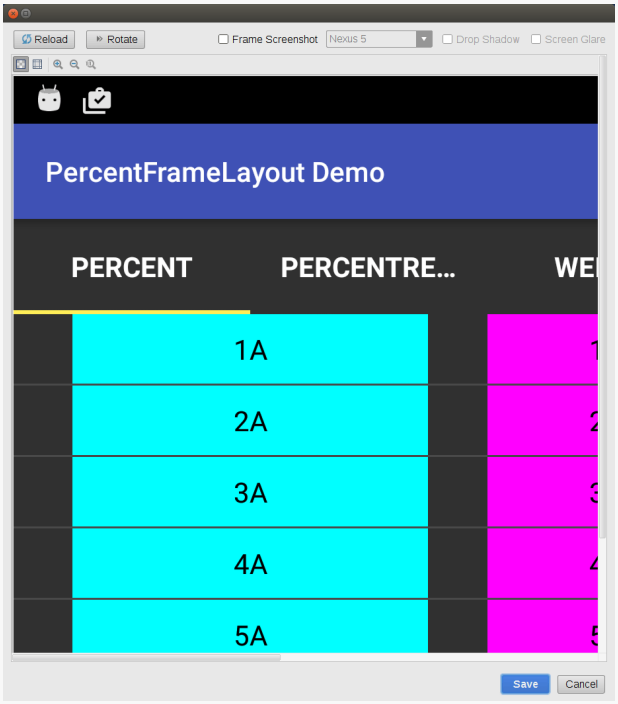


Figure 965: Android Studio Screenshot Dialog

The main area shows the screen at the time you clicked the screenshot toolbar button. Clicking the "Reload" button on the top of the dialog will update the dialog to show the now-current device (or emulator) contents.

Depending on Android Studio version and device characteristics, the dialog may open with the correct orientation. If not, click the "Rotate" button until the image is oriented as you would like it to be.

The "Frame Screenshot" checkbox, if checked, will wrap your screenshot in an image that resembles the hardware from the drop-down list:

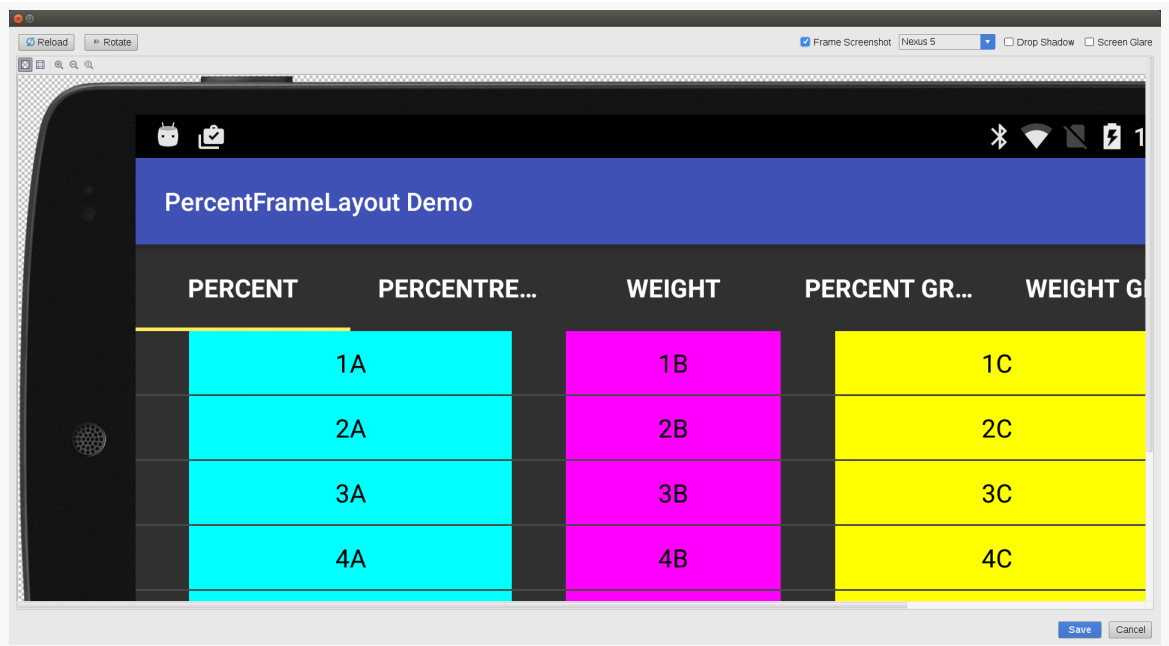


Figure 966: Android Studio Screenshot Dialog, Framing as Nexus 5

The "chessboard" on the outside edges of the image represent transparent areas in the PNG that will be created when you save the image.

Checking the "Drop Shadow" checkbox updates the fake device frame to make it seem like the device is sitting on its edge on some horizontal surface, with a drop-shadow effect. Similarly, checking the "Screen Glare" checkbox adds a fake bit of lighting to the screenshot, as if a light from the upper right side is causing a glare on the fake glass of the fake device frame. Suffice it to say, none of this looks especially realistic.

When you have the screenshot set to your liking, click the "Save" button on the bottom of the dialog, to get a platform-specific "Save As" dialog for you to save your screenshot to wherever you like.

The resulting screenshot will then open in a tab in your IDE. This tab does not let you edit the picture, but it does have an "eyedropper" toolbar button that allows you to examine the image and identify the exact colors of various pixels.

Screencasts

Clicking the second of the two toolbar icons mentioned above brings up a dialog for configuring a screencast:

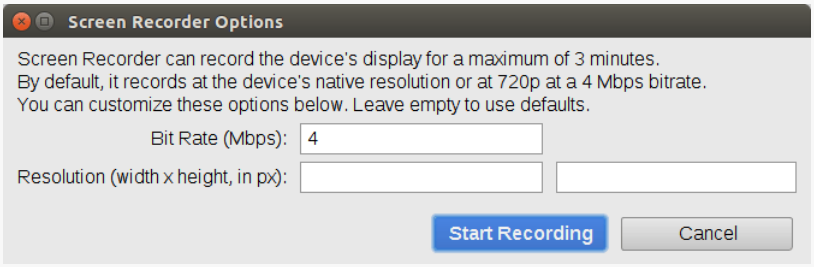


Figure 967: Android Studio Screen Recorder Options Dialog

The particular technique that Android Studio uses to record the screencast is capped at three minutes, which is one of the reasons why there are other alternatives that this chapter will explore.

The bit rate will determine the size of the resulting MP4 file, where a higher bit rate will give you a larger file. However, too low of a bit rate will degrade the quality of the recording, particularly if there is a lot of motion. You will need to experiment for yourself to see what bit rate value works best for you; 4Mbps is the default.

Similarly, normally, the screencast will be at the resolution of the device screen. However, there will be some low-end devices that are incapable of recording a video at that resolution, due to weak video recording support. For those devices, the screencast will be downgraded to 720p. Or, you can attempt to specify the resolution, though you get odd results from the IDE if you try to specify a resolution that is not supported.

Clicking the “Start Recording” button will then start the screencast recording. The dialog that appears has a corresponding “Stop Recording” button. After clicking that, you will be given a “Save As” dialog to save the video wherever you like.

Collecting from the Command Line

The same capabilities that Android Studio taps into to collect screenshots and screencasts graphically are also available to you from the command line, via `adb`. Since `adb` is in the `platform-tools/` directory of your Android SDK installation, if

that directory is in your PATH, you can run adb from any likely directory on your development machine.

Screenshots

adb shell screencap captures a screenshot. This sounds easy enough.

The difficulty is that the screenshot is stored directly on the device or emulator, not on your development machine. This means that taking a screenshot is really a two-step process:

1. Capturing the screen to a PNG on the device
2. Moving that PNG from the device to your development machine

adb shell screencap takes the path for where the PNG should be saved on the device. Since we want to move that PNG to the development machine, it will be simplest if that path is pointing to external storage. What path you use will be tied to what version of Android you are running the screencap command on:

- Android 4.x/5.x: Use /mnt/shell/emulated/0 as the base, which points to the root of external storage
- Android 6.0+: Use /storage/emulated/0 as the base, which points to the root of external storage

You can then use adb pull to copy that PNG to your development machine, followed by adb shell rm to delete the copy that is on the device (to save space, remove clutter, etc.).

For example, the following script would take a screenshot on an Android 6.0 device or emulator and move it to your development machine into whatever the current working directory is:

```
adb shell screencap /storage/emulated/0/screenshot.png
adb pull /storage/emulated/0/screenshot.png .
adb shell rm /storage/emulated/0/screenshot.png
```

Note that the other effects handled by Android Studio, such as rotating the image, are not offered by the command-line interface. Instead, you would use your available image editing tools on your development machine to handle that.

Screencasts

Similarly, `adb shell screenrecord` will record a screencast, saving it as a MP4 file on your device or emulator. And, once again, you will need to use something like `adb pull` to copy that MP4 to your development machine, perhaps followed by `adb shell rm` to remove the copy from the device.

`adb shell screenrecord` is a bit more configurable, though. In addition to the device path to the MP4 file, you can use command-line switches to change the nature of the recording:

- `--size` sets your desired resolution, overriding the default of 1280x720 if your resolution is supported. For example, use `--size 1920x1080` for a 1080p recording.
- `--bit-rate` sets the bit rate, as discussed in the earlier section about screencasts in Android Studio. This is expressed in *bits per second*, so `--bit-rate 8000000` would save at ~8Mbps.
- `--time-limit` will automatically stop the recording after the stipulated number of seconds, capped at a maximum value of three minutes (the equivalent of `--time-limit 180`). Alternatively, while the screencast is recording, press Ctrl-C to stop the recording.

For example, the following script would record a 30-second 1080p screencast on an Android 6.0 device or emulator and move it to your development machine into whatever the current working directory is:

```
adb shell screenrecord --size 1920x1080 --time-limit 30 /storage/emulated/0/
screencast.mp4
adb pull /storage/emulated/0/screencast.mp4 .
adb shell rm /storage/emulated/0/screencast.mp4
```

Collecting from Another App

The three-minute limitation on screencasts, imposed by Android Studio and `adb shell screenrecord`, can be troublesome in some situations.

On Android 5.0 and higher devices, the media projection APIs allow authorized apps to take screenshots and record screencasts. These screencasts do not have an arbitrary time limitation. However, do bear in mind that the videos are stored on the device itself, so disk space can become an issue.

Various apps on the Play Store and elsewhere are available for "out of the box" screencast recording. On the open source front, Jake Wharton wrote and released Telecine, both in a GitHub repository and as an app on the Play Store.

Another chapter in this book shows how you can use the media projection APIs, and one of the sample apps (`andcorder`) can be used akin to how you would use Telecine or `adb shell screenrecorder`.

Tips and Tricks

Note that none of these approaches will record audio along with the video for the screencasts. You will need to use video editing software to add an audio track to the video, whether that comes in the form of a spoken-word voiceover, a soundtrack, or whatever.

While all of the techniques described here will work with devices and emulators, emulators need "Use Host GPU" enabled, at least for API Level 15+ emulators on Linux. Otherwise, your screenshots and screencasts turn out blank.

For screencasts designed to show users how to use an app, you may wish to enable "Show touches" in the Developer Options area of Settings. This will display a white dot where your finger touches the screen, to help illlustrate where you are tapping, sliding, etc. Otherwise, the user may or may not be able to follow exactly what you are doing to cause the app to behave as shown.

ADB Tips and Tricks

Several chapters in this book offer `adb` recipes for doing certain things at the command line. Having the `adb` binary in the `PATH` environment variable for your development machine is very handy, so you can run such commands from anywhere.

However, those other chapters only skim the surface of what sorts of `adb` commands there are and what they can be used for. Several others are presented here.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, and that you know how to work on the command line.

This is the Droid That You Are Looking For

`adb` works well, except when there is more than one visible Android environment, such as two devices, or a device and an emulator. Some commands — notably `adb devices` — work normally. Most other commands will complain that `adb` does not know which of the Android environments the command is supposed to act upon.

There are three switches you can include after `adb` and before the command that control what `adb` will use:

- `-d` says "use the device, there should only be one" (and if there is more than one, you get an error)
- `-e` says "use the emulator, there should only be one" (and if there is more than one, you get an error)
- `-s ...` says "use the environment whose serial number is `...`"

That serial number is the value given in the `adb devices` command. For an actual device, the serial number usually is the real serial number. For an emulator, the serial number is `emulator-NNNN`, where `NNNN` is the value before the AVD name in the title bar of the emulator window. Frequently, that value starts with `5554` and increments by two for each subsequent running emulator. Hence, `-e` is roughly equivalent to `-s emulator-5554`.

Installing and Uninstalling Apps

If you have an APK file that you wish to install — such as the APK edition of this book — you can do that at the command line via `adb install /path/to/the.apk`, where `/path/to/the.apk` is where the APK can be found on your development machine.

If the app already exists on the device or emulator, and you wish to replace it with this new APK, you will have to include the `-r` switch: `adb install -r /path/to/the.apk`. This indicates that you wish to reinstall the app.

`adb install` works if the APK in question was created with an eye towards this sort of manual installation, such as:

- Built using the "Build APK(s)" menu choice in Android Studio
- Built using the `assembleDebug` or `assembleRelease` tasks in Gradle

Starting with Android Studio 3.0, when you run your app from the IDE, while a copy of the APK will left in your module's `build/` directory, that copy has a flag set that blocks it from being installed manually. While `adb install -t` can overcome this block, it is not particularly safe to assume that what Android Studio builds when you run the app from the IDE is safe to be manually installed, let alone distributed anywhere. Use the Gradle tasks or the "Build APK(s)" menu option to create safe APKs instead.

Conversely, `adb uninstall your.application.id` will uninstall the app identified by the application ID (`your.application.id`).

Playing with Permissions

In an `adb install` command, you can include the `-g` switch to proactively grant all of the Android 6.0+ runtime permissions that ordinarily you would need to grant manually.

You can manually grant permissions via the `adb shell pm grant` command. This takes the application ID of your app and the fully-qualified name of the permission:

```
adb shell pm grant com.commonsware.android.perm.tutorial android.permission.CAMERA
```

Similarly, you can use `adb shell pm revoke` to revoke a permission that was already granted to the app:

```
adb shell pm revoke com.commonsware.android.perm.tutorial android.permission.CAMERA
```

These can be useful for testing purposes, either to save you some steps when testing manually, or to blend into automated tests. However, do not become overly reliant upon programmatic permission grants — you need to be sure that your permission flow works for the user, and the user is not going to be using `-g` switches or `adb shell pm grant` commands when using your app.

Starting and Stopping Components

Given an installed app, you can trigger its activities, services, and broadcast receivers from the command line, using `adb shell` to run commands on the device or emulator.

The actual commands are simple:

- `adb shell am start ...` to start an activity
- `adb shell am startservice ...` to start a service
- `adb shell am broadcast ...` to send a broadcast

The challenge is in the `...` part, where you provide command-line switches to construct an `Intent` that will be used for those operations. Here are some common patterns:

- Simple implicit `Intent` with just an action string, use `-a` (e.g., `adb shell am start -a android.intent.action.VOICE_COMMAND`)
- Implicit `Intent` with a `Uri`, use `-a` and `-d` (e.g., `adb shell am start -a android.intent.action.VIEW -d https://commonsware.com`)
- Implicit `Intent` with a different category, use `-a` and `-c` (e.g., `adb shell am start -a android.intent.action.MAIN -c android.intent.category.HOME`)
- Explicit `Intent`: use `-n` (e.g., `adb shell am start -n your.app.id/.YourActivity`)

There are all sorts of command-line switches, for everything from flags to extras, that you can use to build up the `Intent`.

The chapter on the media projection APIs covers a sample screencast recorder, one that can be controlled using these sorts of commands. For example, to start the recording, the `record` shell script from the sample project uses:

```
adb shell am startservice -n
com.commonsware.android.andcorder/.RecorderService -a
com.commonsware.android.andcorder.RECORD
```

This starts the `RecorderService`, using an explicit `Intent` (`-n`) but also providing an action string (`-a`) to state what sort of command we are sending to the service.

Killing Processes and Clearing Data

`adb shell am kill ...` will kill all processes associated with the application ID (`...`).

`adb shell am force-stop ...` will force-stop the app associated with the application ID (`...`), as if the user went into Settings and clicked the "Force Stop" button for the identified app.

`adb shell pm clear ...` will clear the data associated with the application ID (`...`), as if the user went into Settings and clicked the "Clear Data" button for the identified app. This will erase that app's portion of internal storage, plus app-specific directories on external storage (e.g., `getExternalFilesDir()`).

Changing Display Metrics

One reason why developers use emulators is because they lack hardware for device scenarios that they wish to test. Two such scenarios are screen size and density. Many developers have only a device or two to test against, and they may need to try out screen sizes and densities that their hardware does not offer directly.

However, if you have a device with a *higher* resolution or density, you can use `adb` to have the device fake operating as a lower-resolution or lower-density device.

Specifically, on Android 4.3 and higher, `adb shell wm size 1280x800` would tell an Android device to pretend to have a WXGA800 display. You will see the smaller area centered within the overall device screen.

Note, though, that the device may no longer honor orientation changes by rotating the device. You will need to stipulate your size based upon the orientation that you are holding the device and the default orientation of the device itself.

For example, running the above command on a Nexus 9 gives you the following, regardless of whether the Nexus 9 is in portrait or landscape:

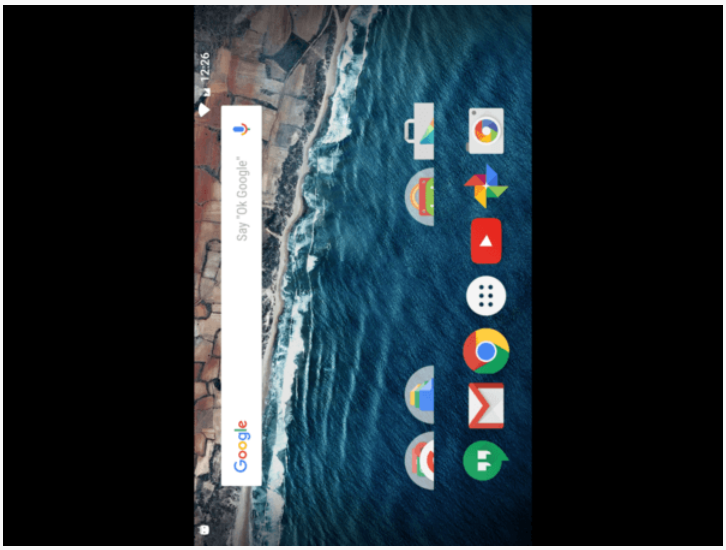


Figure 968: 1280x800 Display Size, On a Nexus 9, Held in Landscape

If you were planning on testing the Nexus 9 in portrait mode and wanted a landscape WXGA800 display, this is fine. More likely, you will need to change the order of your dimensions in the command. So, running `adb shell wm size 800x1280` gives you:

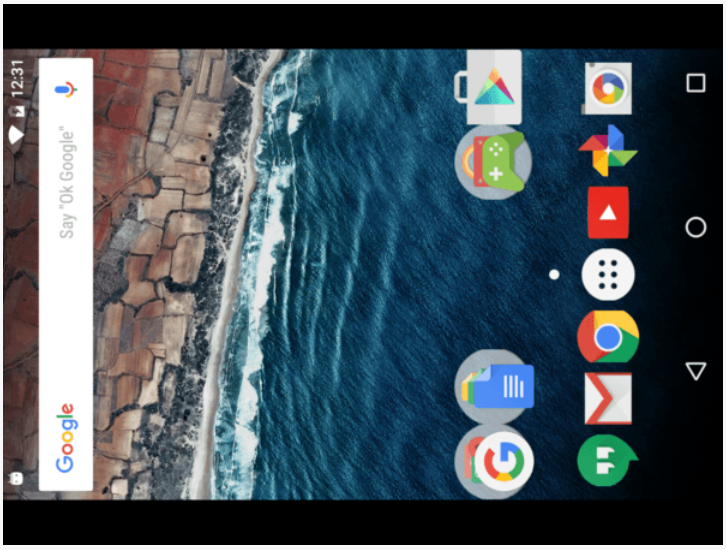


Figure 969: 800x1280 Display Size, On a Nexus 9, Held in Landscape

Here, at least, the device orientation matches the reduced-size screen orientation, if you were to hold the device in portrait mode.

If you prefer, you can use `dp` units instead, by appending `dp` after the values. Using `reset` instead of a resolution will return the device to its native resolution.

Similarly, `adb shell wm density 160` will have the device behave as though it has 160dpi screen density. This, however, starts to look a little strange, and you may find it difficult to completely understand what is going on, at least with third-party apps like the home screen. `adb shell wm density reset` returns the device to its natural screen density.

Stetho

In 2015, Facebook announced Stetho, "a new debugging platform for Android". That description is more apt than you might think, in that Stetho allows you to examine your view hierarchy, see network requests, and otherwise analyze your project... using Chrome.

Wait, Wut? Chrome?

Many modern Web browsers have Web client debugging tools, either built into the browser itself or available as an extension or other add-on. These tools can let you browse the content of the Web page, see network requests, and otherwise analyze the content of a browser tab.

Stetho leverages the Chrome Developer Tools, available in Chrome and Chromium, to have those tools examine an Android app, rather than a browser tab.

This works by way of Chrome Developer Tools' support for remote debugging. Stetho basically embeds a small server in your app that speaks the same protocol that Chrome Developer Tools uses for remote debugging. From Chrome's perspective, your Android app is just another Web browser. In reality, Stetho translates Chrome Developer Tools' requests (e.g., "give me your DOM") into things that would help an Android developer (e.g., "give me your view hierarchy").

Basic Stetho Integration

Stetho is not that hard to integrate, though Facebook's documentation for it would have you ship Stetho in production in a `release` build, which is not an especially good idea.

The Diagnostics/Stetho sample project will show you how to hook Stetho up to your app, in a way that allows you to ship Stetho only in your debug builds. Overall, this app is yet another variant on the "show the latest android questions from Stack Overflow" introduced originally in the chapter on Internet access. This particular one uses Retrofit 2.x and Picasso for the network requests... though as you will see, we have both of those libraries delegate the actual network I/O to OkHttp 3.x.

Adding the Stetho Dependency

The official Stetho dependency is com.facebook.stetho:stetho, for some version (e.g., 1.4.2). However, if you want to allow Stetho to help you debug your network requests, you will need a different dependency, based on which HTTP client API you are using for those network requests:

HTTP Client API	Stetho Dependency
OkHttp	com.facebook.stetho:stetho-okhttp
OkHttp 3.x	com.facebook.stetho:stetho-okhttp3
HttpURLConnection	com.facebook.stetho:stetho-urlconnection

Those each have com.facebook.stetho:stetho as a transitive dependency, so you only need one Stetho dependency, based on your HTTP client API.

However, there are two issues:

1. There is a bug in Stetho 1.4.2, where they forgot a particular dependency on part of the Android Support libraries
2. Stetho 1.4.2 has a transitive dependency on appcompat-v7, for no obvious reason

So, the app/build.gradle file from the Stetho sample project cleans that up:

```
dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
    implementation 'com.squareup.okhttp3:okhttp:3.8.0'
    implementation 'com.jakewharton.picasso:picasso2-okhttp3-downloader:1.0.2'
    debugCompile('com.facebook.stetho:stetho-okhttp3:1.4.2') {
        exclude group: 'com.android.support', module: 'appcompat-v7'
    }
    debugCompile 'com.android.support:support-compat:25.2.0'
}
```

(from Diagnostics/Stetho/app/build.gradle)

Specifically, we are using the `stetho-okhttp3` dependency, so we get Stetho plus its OkHttp 3.x support.

We use `debugCompile` so that these libraries are only used in `debug` builds, not `release` builds. We block the unnecessary `appcompat-v7` transitive dependency via a Gradle `exclude` directive, while adding a `debugCompile` statement for `support-compat`, a part of the `support-v4` library group that contains the missing support classes.

If you use Stetho in a project that is using `appcompat-v7` itself, you might still want to use the `exclude` directive when pulling in Stetho, to help ensure that you get the version of `appcompat-v7` that *you* want, not the one that Stetho wants.

Creating a Debug Sourceset

Ideally, as much of our Stetho-specific stuff as possible should *not* be in the `main` source set, as that is what ships to customers. Using `debugCompile` keeps the Stetho dependencies out of a `release` build, but we are going to need some code to initialize and configure Stetho. That code, ideally, goes somewhere other than `main`.

As was covered in the chapter on the Gradle project structure, we can have a `debug` source set, as a peer of `main`. Everything in `debug` will be merged into our project for a `debug` build but will be ignored with a `release` build. So, other than the `debugCompile` statements in `app/build.gradle`, the rest of our Stetho stuff will go into `debug`.

Adding the Stetho Application

Stetho requires some initialization work, and Facebook recommends that this be done in a custom `Application` object. This is a process-wide singleton, initialized when our process is forked, and so it is good for one-time, process-scope initialization work.

However, we really want this to be in the `debug` source set, and that requires a little bit of work.

The Main Application

Over in the `main` source set, we have an `App` class that extends `Application` and provides initialization for all build types:

```
package com.commonsware.android.stetho;

import android.app.Application;
import okhttp3.OkHttpClient;

public class App extends Application {
  private OkHttpClient ok;

  @Override
  public void onCreate() {
    super.onCreate();

    ok=buildOkBuilder().build();
  }

  OkHttpClient getOk() {
    return(ok);
  }

  protected OkHttpClient.Builder buildOkBuilder() {
    return(new OkHttpClient.Builder());
  }
}
```

(from Diagnostics/Stetho/app/src/main/java/com/commonsware/android/stetho/App.java)

Here, we are initializing an app-wide instance of `OkHttpClient`, to be used for our network requests. In other incarnations of this sample app, we either create the `OkHttpClient` directly in `QuestionsFragment` (where the network I/O is triggered) or are not using OkHttp at all. Here, we are doing this at the process level, for two reasons:

1. For a more complex app, where you are doing network I/O in several places, you may want a single `OkHttpClient` instance with all of your configuration.
2. Stetho needs access to the `OkHttpClient`, if we want it to report on network access. And, we need to use that same Stetho-configured `OkHttpClient` for all our network access that we want Stetho to report.

Note that we have a `protected` method, `buildOkBuilder()`, that sets up the `OkHttpClient.Builder` that we use to create the `OkHttpClient` instance. We will see that method again shortly... in a Stetho-specific subclass.

The `main` edition of the manifest then says that we should use `App` by setting `android:name` on `<application>`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.stetho"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.INTERNET"/>

  <application
    android:allowBackup="false"
    android:name="com.commonsware.android.stetho.App"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name="com.commonsware.android.stetho.MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from Diagnostics/Stetho/app/src/main/AndroidManifest.xml)

This causes Android to create an instance of `App`, instead of the standard `Application` class, as the process-wide singleton.

The Debug Application

Over in the `debug` source set, we have a `StethoApp` class that extends the `App` class from the `main` source set:

```
package com.commonsware.android.stetho;

import com.facebook.stetho.Stetho;
import com.facebook.stetho.okhttp3.StethoInterceptor;
import okhttp3.OkHttpClient;

public class StethoApp extends App {
  @Override
  public void onCreate() {
    super.onCreate();

    Stetho.initializeWithDefaults(this);
```

```
  }

  @Override
  protected OkHttpClient.Builder buildOkBuilder() {
    return(super.buildOkBuilder().addNetworkInterceptor(new StethoInterceptor()));
  }
}
```

(from Diagnostics/Stetho/app/src/debug/java/com/commonsware/android/stetho/StethoApp.java)

debug source set classes can "see" those in main, which is why we can successfully subclass App.

Here, in onCreate(), we initialize Stetho with a default configuration, using initializeWithDefaults(). If all we wanted was basic Stetho integration, without network call tracking, this would be all that we need.

To integrate network tracking, you need some additional code, based on the particular HTTP client API that you are using. We pulled in the stetho-okhttp3 dependency and are using OkHttp3, so we need to add an OkHttp network interceptor. Such an interceptor is called on each network request, for cross-cutting concerns like logging.

So, we override buildOkBuilder(), call addNetworkInterceptor() to add a StethoInterceptor to the interceptor chain, and return the modified OkHttpClient.Builder. Now, when App uses buildOkBuilder(), it will pull in our subclass override... if the Application singleton for our process is a StethoApp, instead of an App.

Overriding the Application

That requires us to teach a manifest to use StethoApp, in the same way that we modified the main source set's manifest to use App. The debug source set can have its own manifest, and that manifest can override certain settings from main:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  package="com.commonsware.android.stetho"
  android:versionCode="1"
  android:versionName="1.0">

  <application
    android:name="com.commonsware.android.stetho.StethoApp"
    tools:replace="android:name">
```

```
  </application>

</manifest>
```

(from Diagnostics/Stetho/app/src/debug/AndroidManifest.xml)

Here we tell Android to use `StethoApp` for the singleton (via the same `android:name` attribute on the `<application>` element). And, via `tools:replace`, we tell the build tools to use *our* definition for `android:name`. Otherwise, the build will fail, as there is a conflict between what the `main` manifest has (`App`) and what the `debug` manifest has (`StethoApp`).

The "Merged Manifest" tab in Android Studio shows that our resulting `android:name` attribute for a `debug` build uses `StethoApp`:

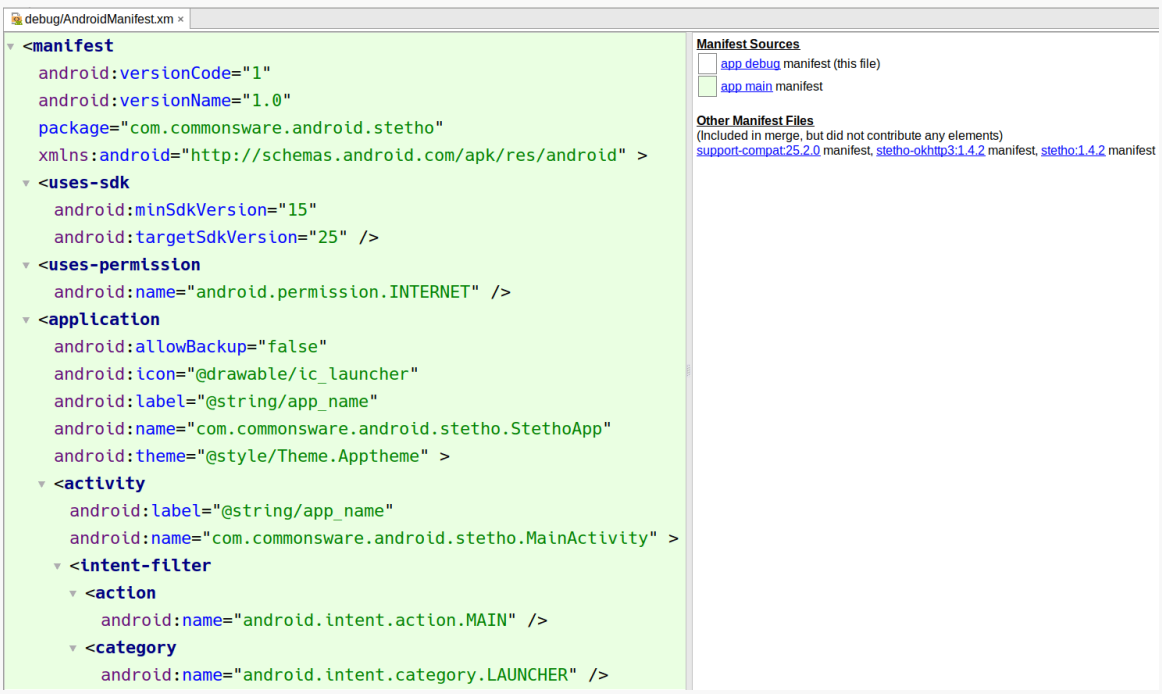


Figure 970: Merged Manifest for Debug Build, Showing StethoApp

If we use Android Studio's Build Variants view to switch to the `release` build, now the merged manifest shows that the regular `App` class will be used:

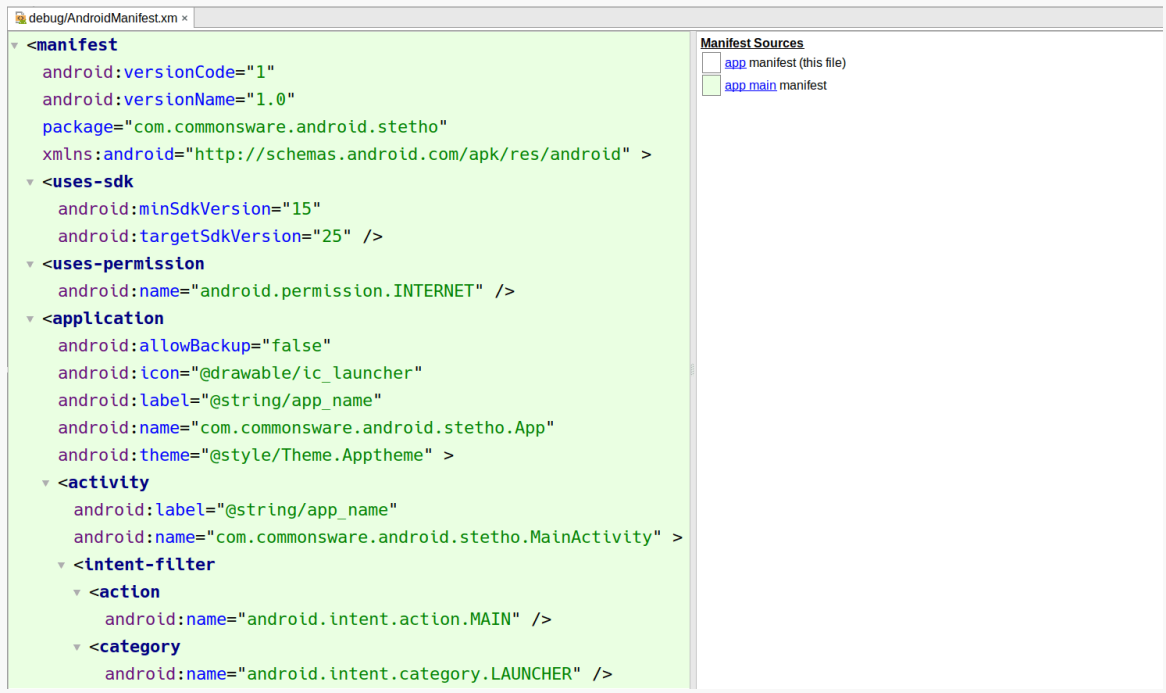


Figure 971: Merged Manifest for Release Build, Showing No StethoApp

So, we have achieved our objective: use Stetho in `debug` builds but not in `release` builds.

So, with all that, let's see what Stetho actually gives us.

Connecting Chrome to Your App

With your app running on an emulator, or on a device connected to your development machine, open up Chrome or Chromium and visit `chrome://inspect/#devices`. This should bring up a page that shows you the available "remote targets", which should include your Stetho-enabled app:

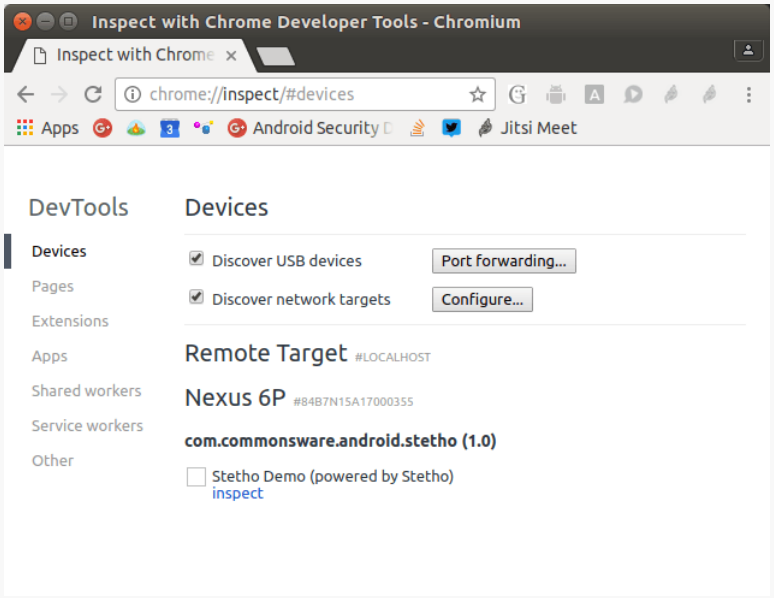


Figure 972: Chrome Developer Tools, Showing Remote Targets

While a square that resembles an unchecked checkbox appears next to your app, that seems to have no use. Instead, click the "inspect" link below your app to bring up the Chrome Developer Tools on your Stetho-enabled app.

What You Get In Chrome Dev Tools

Not every tool in Chrome Developer Tools is populated by Stetho. So, for example, the "Sources" and "Timeline" tools will remain empty. However, some of the other tools will give you insights into your app, courtesy of Stetho:

Elements: View Hierarchy

The "Elements" tool is roughly comparable to Android Studio's Layout Inspector or the `uiautomator` tool. It allows you to examine the view hierarchy of your UI.

However, since the Elements tool is designed for examining HTML and Web pages, the view hierarchy is represented in the form of pseudo-HTML elements:

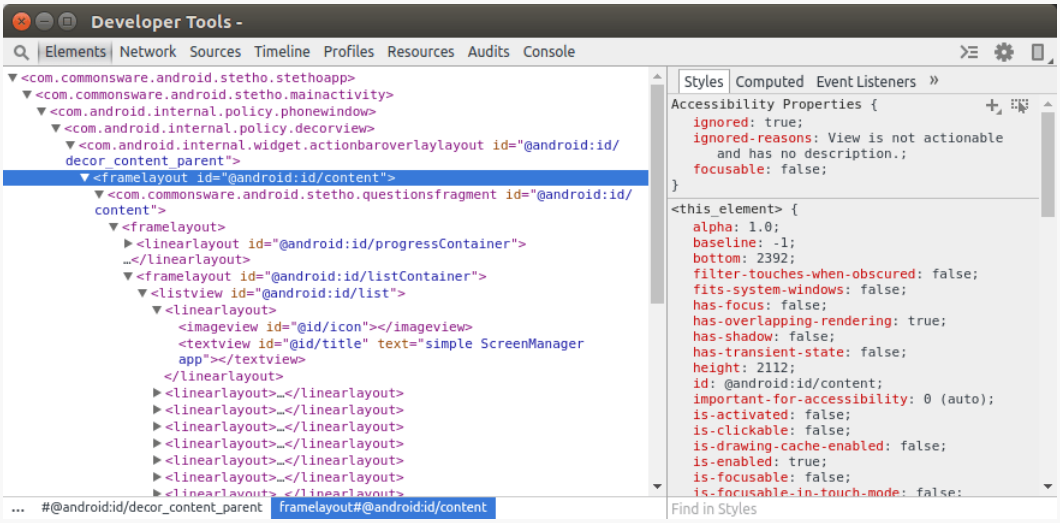


Figure 973: Elements Tool, Showing Stetho Sample UI Elements

The rules for HTML elements are akin to the rules for layout XML resources:

- If the view is from a well-known package, like `android.widget`, the HTML element uses the bare class name
- Otherwise, the HTML element uses the fully-qualified class name

However, these are converted into all lowercase, since HTML elements are not case-sensitive. This takes a bit of getting used to.

The Styles tab on the right will show common properties of the highlighted view, represented as if they were CSS styles (e.g., `is-enabled`). The rightward-pointing caret in tabs on the right has a “Properties” option. Selecting will give you a dump of all of the fields inside of the highlighted view, presumably obtained via Java reflection APIs:

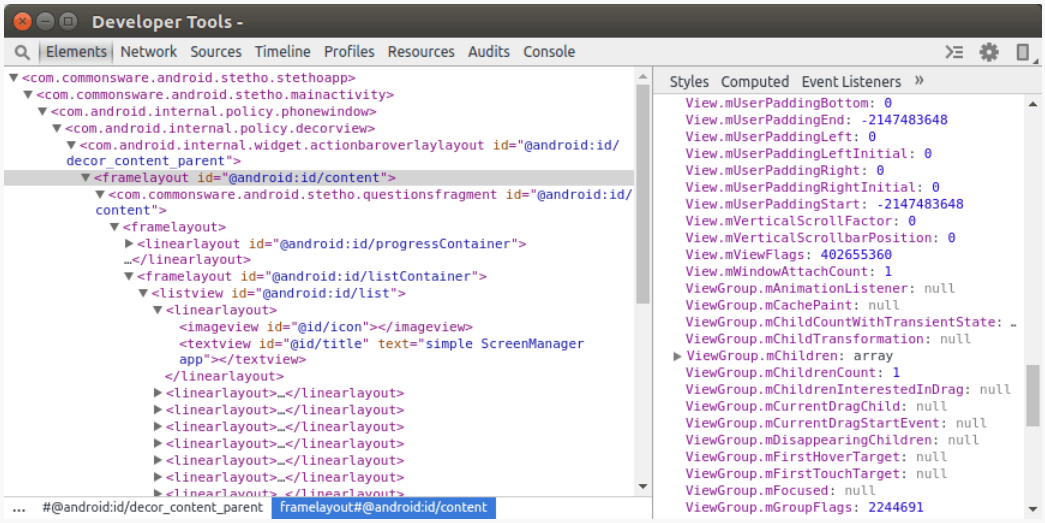


Figure 974: Elements Tool, Showing Properties of a Stetho Sample UI Element

Also, on your Android device or emulator, the view that you select in the Elements tool gets a tint applied to it, akin to how the Layout Inspector and `uiautomatorviewer` tint the regions of the screenshot shown in each of those tools. This helps you to identify exactly what widget or container the highlighted element refers to.

Network: HTTP Requests

The "Network" tool, for a standard Web page, shows all of the HTTP requests that were made in support of rendering that Web page. With Stetho, that tool shows the HTTP requests made by your app... that went through whatever API you configured when you set up Stetho. In the sample app, we configured a particular `OkHttpClient` to use a Stetho-supplied network interceptor. So, if we click the refresh action bar item — forcing a fresh set of network calls – we will see those in the network tool:

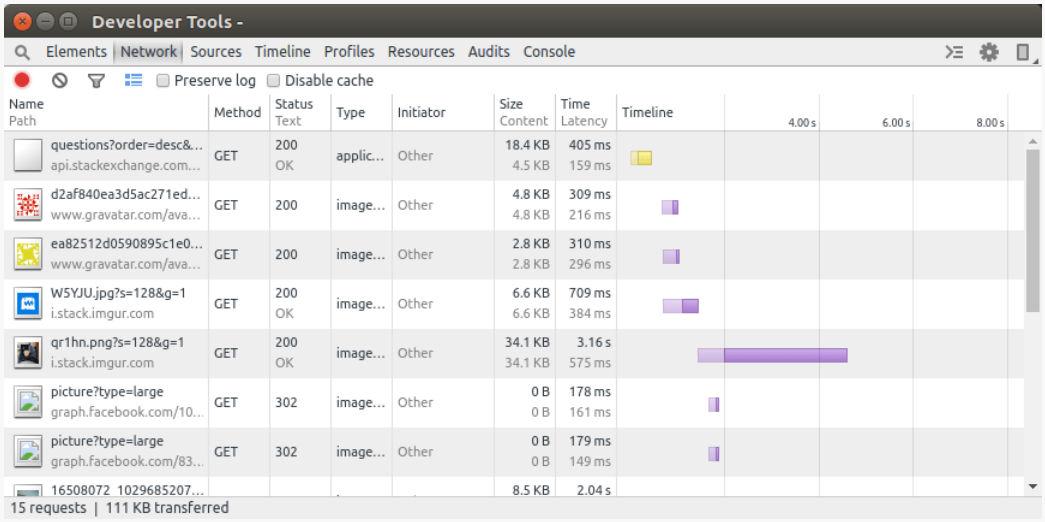


Figure 975: Network Tool, Showing HTTP Requests from Sample App Refresh

The URLs shown in the table are clickable. Clicking one opens up a set of tabs on the side, with a "Headers" tab open by default, to show you the HTTP headers of the request and response:

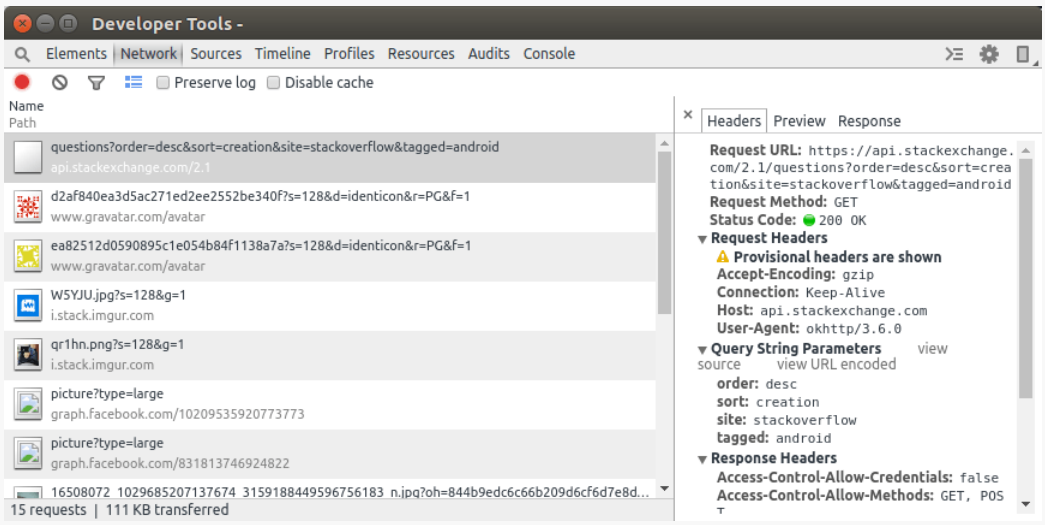


Figure 976: Network Tool, Showing HTTP Request and Response Headers

The "Preview" tab will show the response, using a structure associated with the MIME type. So, for example, the Web service call made to the Stack Exchange API gives us a tree representation of the JSON response:

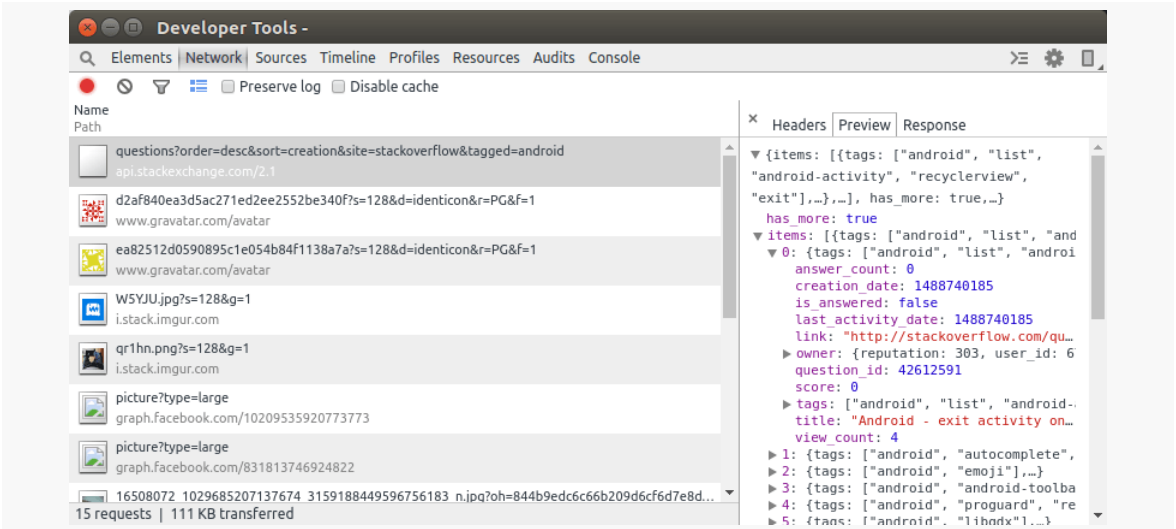


Figure 977: Network Tool, Showing JSON Response

...whereas the response shown for one of the images is the image itself, along with some key details (e.g., size):

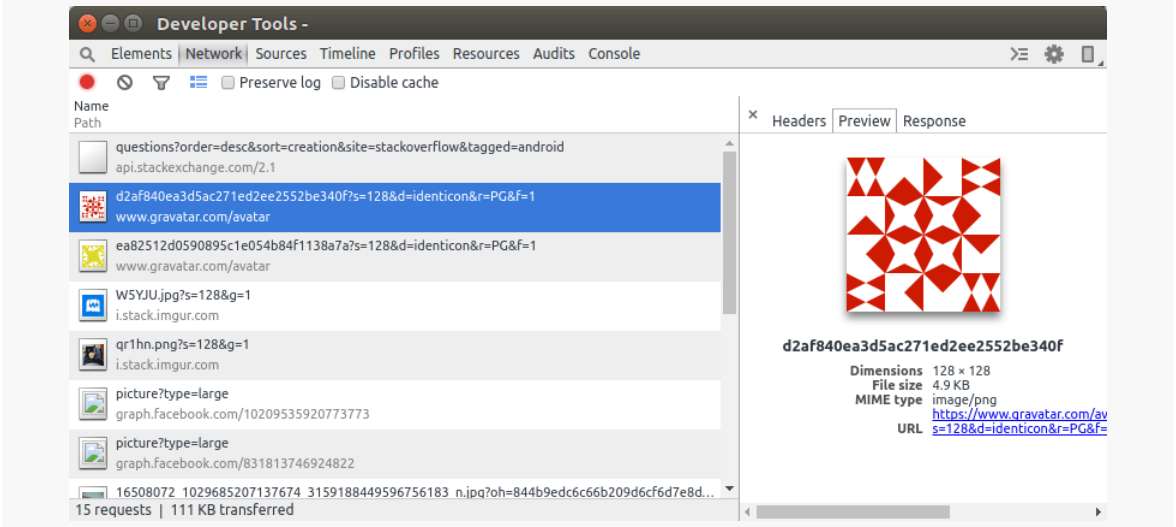


Figure 978: Network Tool, Showing Image Response

Screencast: Your UI, Mostly

The button in the upper-right of the Dev Tools window, that looks like a phone, will open up the “Screencast” pane:

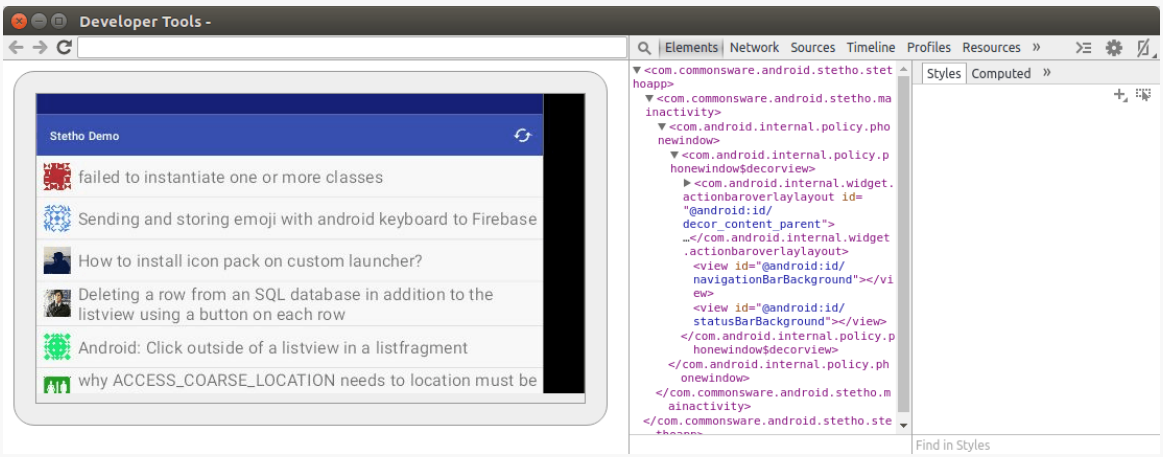


Figure 979: Stetho Screencast

This shows a semi-live edition of your UI. It has occasional hiccups, particularly with scrollable content, but generally works.

Resources: In-Place Database CLI

The Resources tool gives you access to your SQLite databases, assuming that they are stored in the default location used by `SQLiteOpenHelper`.

Alas, our earlier sample app has no database.

The `Diagnostics/StethoDB` sample project is a clone of one of the database samples, where we are storing some gravity constants culled from `SensorManager` in a database table as starter data, and we allow the user to add more constants.

This time, though, we add the same basic `debug/` source set as was used in the `Diagnostics/Stetho` sample. In this case, the `StethoApplication` extends `Application` (as the main app has no custom `Application` class) and only initializes Stetho itself:

```
package com.commonsware.android.stetho;

import android.app.Application;
import com.facebook.stetho.Stetho;

public class StethoApp extends Application {
  @Override
  public void onCreate() {
```

```
    super.onCreate();

    Stetho.initializeWithDefaults(this);
  }
}
```

(from Diagnostics/StethoDB/app/src/debug/java/com/commonsware/android/stetho/StethoApp.java)

We also have two different Stetho dependencies. One is just `com.facebook.stetho:stetho`, which is the base Stetho artifact. We also load in `com.facebook.stetho:stetho-js-rhino`, which enables the JavaScript console, as we will see later in this chapter. Since this sample app is not doing any network I/O — let alone with OkHttp3 — we do not need the `com.facebook.stetho:stetho-okhttp3` artifact.

In the Resources tool, you can expand the "Web SQL" tree to see all of your databases. Expanding a database brings up the tables in that database, and clicking on a table shows you its current contents:

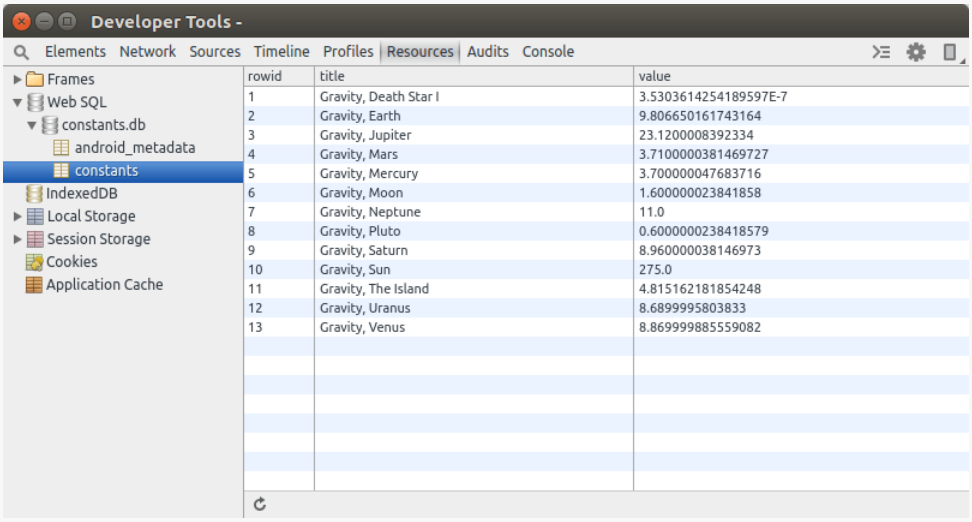


Figure 980: Resources Tool, Showing Constants Table Content

Clicking on the database name itself (e.g., `constants.db`) in the tree brings up an interactive SQL utility, akin to the `sqlite3` command-line tool. You can execute arbitrary SQL statements and see their results:

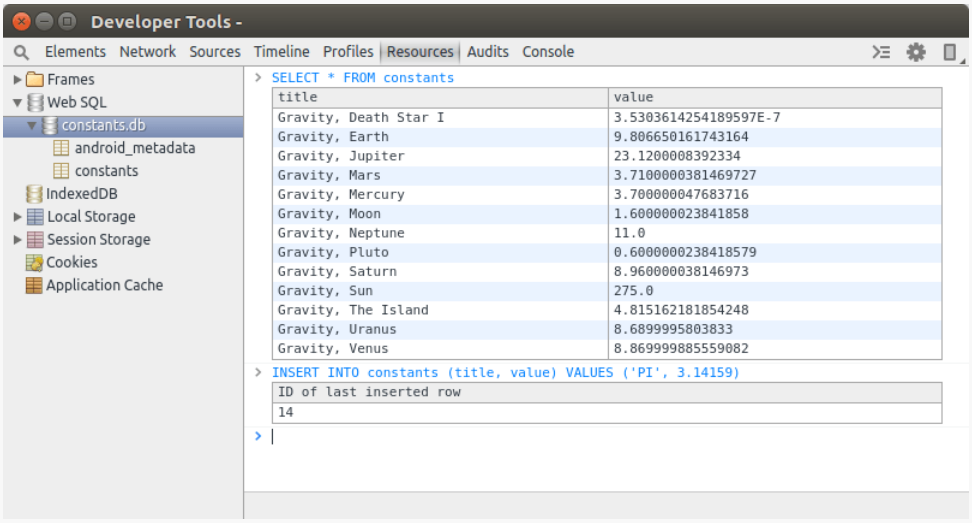


Figure 981: Resources Tool, Showing Interactive SQL

This is all using your live database on the device or emulator. This is much more convenient than using `adb shell run-as` commands to pull a SQLite database off of production hardware and then opening it in some other SQLite utility.

Console: Example Environment via JavaScript

The `com.facebook.stetho:stetho-js-rhino` is completely optional. And, if you add it, it will expand your APK by a fair bit (~1MB and over 6,000 methods). However, it adds an interesting feature: JavaScript access to your `Application` object, via the Dev Tools' Console.

The `context` synthetic JavaScript global object is your custom `Application`, and you can interact with it using the Rhino JavaScript-on-Java interpreter. The return value of any JavaScript expression will be whatever the Java code returns, or `"undefined"` for `void` methods.

Just typing in `context` allows you to inspect the contents of your custom `Application`:

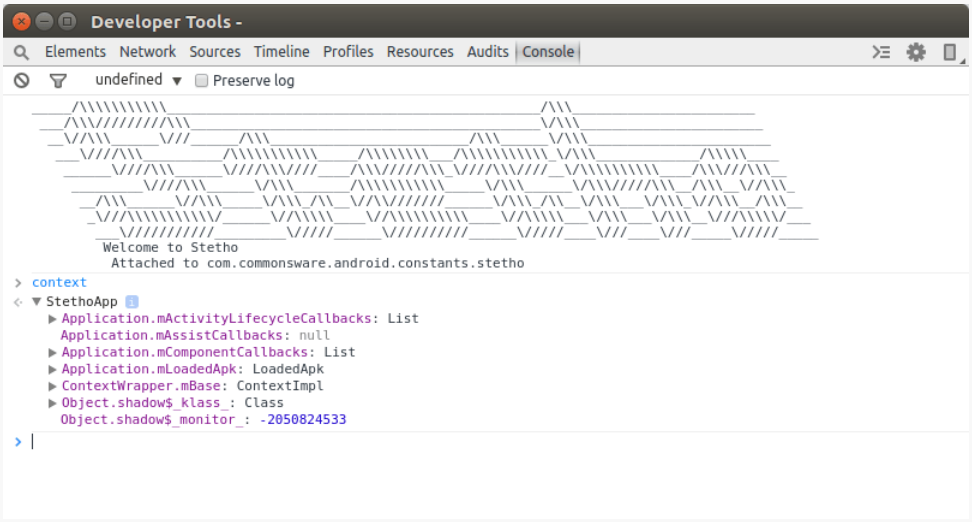


Figure 982: Console Tool, Inspecting the Application Context

`importPackage()` works akin to how `import` does in traditional Java code, to allow you to reference classes from other packages. Anything that can be reached via your custom `Application`, or via some `static` field, can be done, such as sending a broadcast:

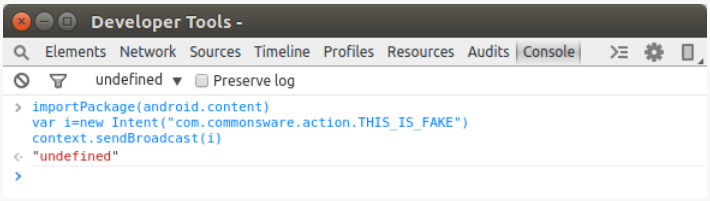


Figure 983: Console Tool, Sending a Broadcast

Note: use Shift-Enter to write multiple lines of JavaScript code without executing them, until you press Enter after the last line.

Getting Help with Stetho

The Stetho Web site and its corresponding GitHub repository appear to be your primary places for getting assistance with Stetho.

Trail: Tuning Android Applications

Issues with Speed

Mobile devices are never fast enough. Either they are slow in general (e.g., slow CPU) or they are slow for particular operations (e.g., advanced game graphics).

What you do not want is for your application to be unnecessarily slow, where the user determines what is and is not "necessary". Your opinion of what is "necessary", alas, is of secondary importance.

This part of the book will focus on speed, including how you can measure and reduce lag in your applications. First, though, let's take a look at some of the specific issues surrounding speed.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Getting Things Done

In some cases, you simply cannot seem to get the work done that you want to accomplish. Your database query seems slow. Your encryption algorithm seems slow. Your image processing logic seems slow. And so on.

The limits of the device will certainly make this more of a problem than it might otherwise be. Even a current-era multi-core device will be slow compared to your average notebook or desktop, as mobile CPUs cannot readily be directly compared to desktop and notebook CPUs. Also, this sort of speed issue is pervasive throughout

computing, with decades of experience to help developers learn how to write leaner code.

This part of the book will aim to help you identify where the problem spots are, so you know what needs optimization, and then some Android-specific techniques for trying to improve matters.

Your UI Seems… Janky

Sometimes, the speed would be less of an issue for the user, if it was not freezing the UI or otherwise making it appear sluggish and "janky".

The Android widget framework operates in a single-threaded mode. All UI changes — from setting the text of a `TextView` to handling scrolling of a `GridView` — are processed as events on an event queue by the main application thread. That same thread is used for most UI callbacks, including activity lifecycle methods (e.g., `onCreate()`) and UI event methods (e.g., `onClick()` of a `Button`, `getView()` of an `Adapter`). Any time you take in those methods on the main application thread tie up that thread, preventing it from processing other GUI events or dispatching user input. For example, if your `getView()` processing in an `Adapter` takes too long, scrolling a `ListView` may appear slow compared to other `ListView` widgets in other applications.

Your objective is to identify where things are slow and move them into background operations. Some of this has been advised since the early days of Android, such as moving all network I/O to background threads. Lots of work has gone into providing libraries for you to be able to easily move common tasks, like loading images, onto background threads.

This part of the book will point out ways for you to find out where you may be doing unfortunate things on the main application thread and techniques for getting that work handled by a background thread, or possibly eliminated outright.

Not Far Enough in the Background

Sometimes, even work you are trying to do in the background will seem to impact the foreground.

For example, you might think that your `Service` is automatically in the background. An `IntentService` does indeed use a background thread for processing commands

via `onHandleIntent()`. However, all lifecycle methods of any `Service`, including `onStartCommand()`, are called on the main application thread. Hence, any time you take in those lifecycle methods will steal time away from GUI processing for the main application thread. The same holds true for `onReceive()` of a `BroadcastReceiver` and all the main methods of a `ContentProvider` (e.g., `query()`).

Even your background threads may not be sufficiently in the background. A process runs with a certain priority, using Linux process management APIs, based upon its state (e.g., if there is an activity in the foreground, it runs at a higher priority than if the process solely hosts some service). This will help to cap the CPU utilization of the background work, but only to a point. Similarly, threads that you fork — directly or via something like `IntentService` — may run at default priority rather than a lower priority. Even with lower priorities for the thread or process, every CPU instruction executed in the background is one clock tick that cannot be utilized by the foreground.

This part of the book will help you identify where you are taking lots of time on various threads and will help you manually manage priorities to help minimize the foreground impact of those threads, in addition to helping you reduce the amount of work those threads have to do.

Playing with Speed

Games, more so than most other applications, are highly speed-dependent. Everyone is seeking the "holy grail" of 60 frames per second (FPS) necessary for smooth animated effects. Not achieving that frame rate overall may mean the application will not appear quite as smooth; sporadically falling below that frame rate will result in jerky animation effects, much like the "janky" UIs in a non-game Android application.

For example, a classic problem with Android game development is garbage collection (GC). The original Android garbage collector was a "stop the world" implementation, that would freeze the game long enough for a bit of GC work to be done before the game could continue. This behavior pretty much guaranteed sporadic failures to maintain a consistent frame rate. This caused game developers to have to take particular steps to avoid generating any garbage, such as maintaining its own object pools, to minimize or eliminate garbage collection pauses. While Android 2.3 and beyond have taken steps to have garbage collection be more concurrent, there are still short pauses (1-2ms, typically), where all threads have to be suspended to wrap up the GC run.

This book does not focus much on specific issues related to game development, though many of the techniques outlined here will be relevant for game developers.

Finding CPU Bottlenecks

CPU issues tend to manifest themselves in three ways:

- The user has a bad experience when using your app directly — scrolling is sluggish, activities take too long to display, etc.
- The user has a bad experience when your app is running in the background, such as having slower frame rates on their favorite game because you are doing something complex in a service
- The user has poor battery performance, driven by your excessive CPU utilization

Regardless of how the issue appears to the user, in the end, it is a matter of you using too much CPU time. That could be simply because your application is written to be constantly active (e.g., you have an everlasting service that uses `TimerTask` to wake up every second and do something). There is little anyone can do to help that short of totally rethinking the app's architecture (e.g., switch to `AlarmManager` and allow the user to configure the polling period).

However, in many cases, the problem is that you are using algorithms – yours or ones built into Android — that simply take too long when used improperly. This chapter will help you identify these bottlenecks, so you know what portions of your code need to be optimized in general or apply the techniques described in later chapters of this part of the book.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

Android Studio's Profiler

In Android Studio, the Android Profiler tool allow you to examine the real-time behavior of your app with respect to various system resources, such as the CPU.

Opening the Profiler

If one or more debuggable processes are running on an attached device or an emulator, the Android Profiler tool will appear as an option on the bottom dock:

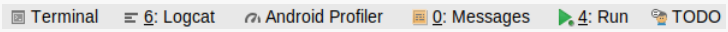


Figure 984: Android Studio Bottom Tool Dock, Showing Android Profiler

Opening the profiler will give you a real-time look at the consumption of CPU, heap space, and network associated with your process:

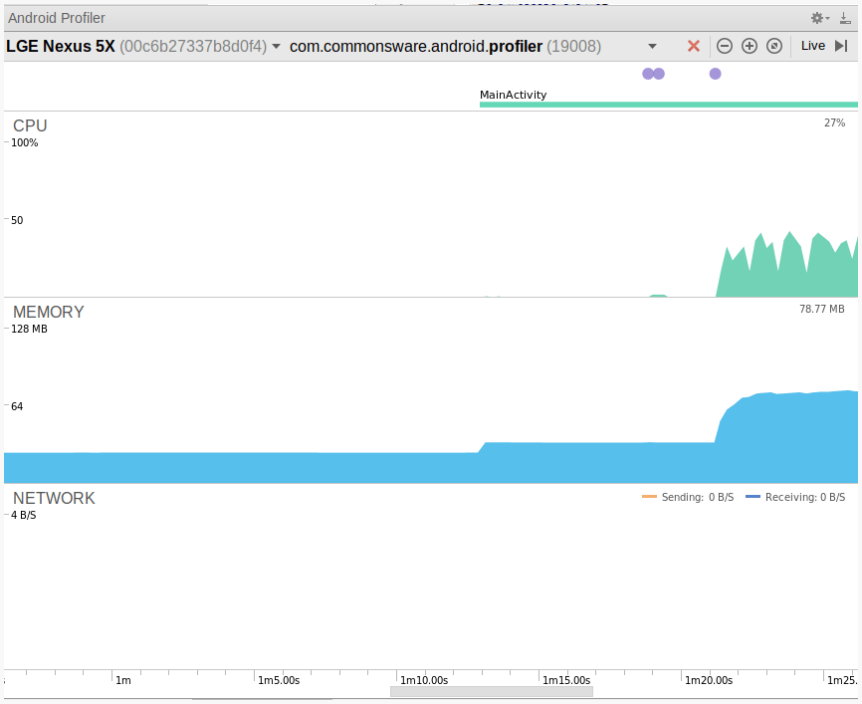


Figure 985: Android Profiler

The top area will show a green bar when your UI is in the foreground, with a note as to what activity of yours is in the foreground. Purple dots show when you have received user input, such as taps on buttons.

Manipulating the Profiler

Across the top, there are:

- Drop-downs to choose the device and process that you are profiling
- A close button to close this tool
- Buttons to zoom in, zoom out, or reset the zoom to its default level
- A "Live" button that serves to pause the results or, from a paused state, resume normal real-time output

Hovering your mouse over a portion of the graph will show you the specific value at that point in time for the particular metric you are hovering over:

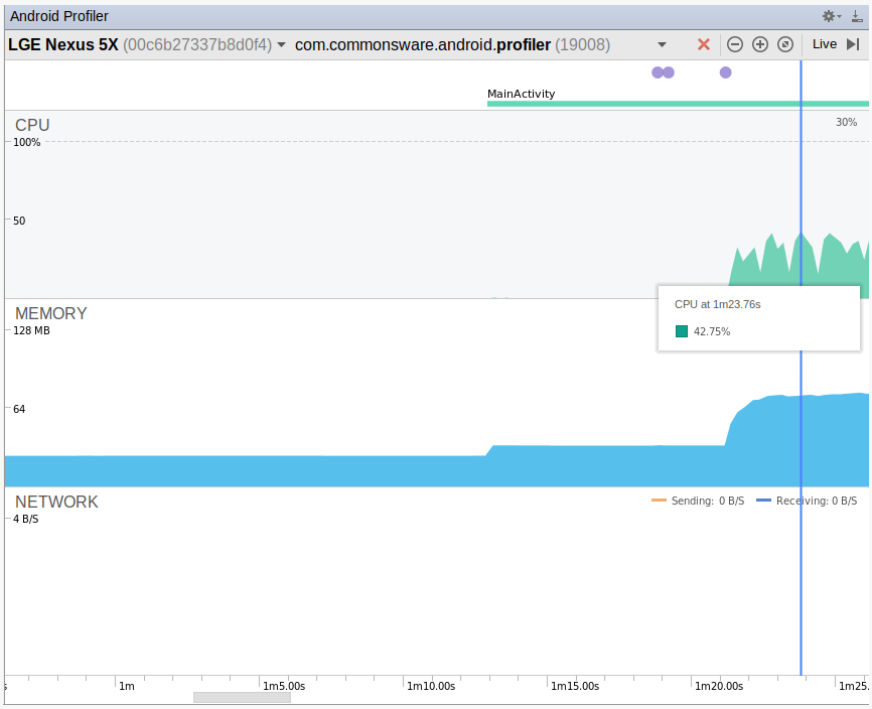


Figure 986: Android Profiler, Showing Specific CPU Usage

Clicking on the CPU graph at any point shows you more details about the per-thread CPU utilization:

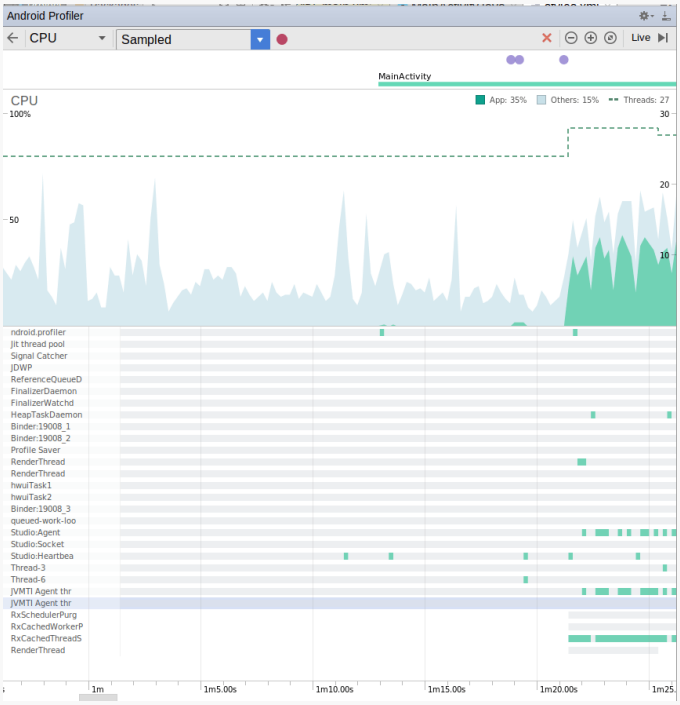


Figure 987: Android Profiler, Showing Per-Thread CPU Usage

Hey, Why Is My App Running So Slow Now?

Android Studio 3.0's Profiler has issues. Depending on the nature of your app, you may find that the profiler dramatically slows down your app.

Use the red X button to close the profiler, and your app should return to normal.

Method Tracing

The #1 tool in your toolbox for finding out where bottlenecks are occurring in your application is method tracing. This will record your code and how long it takes your various methods to do their work. You can use this to look for outliers:

- Methods that are called way too frequently
- Methods that call other methods way too frequently
- Methods that take a lot of time in their own statements, including things like blocking on I/O

OK, What Is Method Tracing, Really?

Technically, the method tracing in Android is performed by the virtual machine, under the direction of either your IDE or requests from your application code. Dalvik or ART will write the "trace data" (call graphs showing methods, what they call, and the amount of time in each) to a file on external storage of the device or emulator. Your IDE then views these trace files in a GUI, allowing you to visualize "hot spots", drill down to find where the time is being taken, and so forth.

At the time of this writing, method tracing is designed for use on single-core devices. Results on multi-core devices may be difficult to interpret.

Collecting Trace Data

Hence, the first step for finding where your CPU bottlenecks lie comes in the form of collecting trace data for analysis. As mentioned, there are two approaches for requesting trace data be logged: using the `Debug` class, and using your IDE.

Debug Class

If you know what chunk of code you want to profile, one way to arrange for the profile is to call `startMethodTracing()` on the `Debug` class. This takes the name of a trace file as a parameter and will begin recording all activity to that file, stored in the root of your external storage. You need to call `stopMethodTracing()` at some point to stop the trace — failing to do so will leave you with a corrupt trace file in the end.

Note that your application will need the `WRITE_EXTERNAL_STORAGE` permission for this to work. If your application does not normally need this permission, make yourself a note to remove it before you ship the production edition of your product, as there is no sense asking for any more permissions than you absolutely need. Or, put this permission in a `debug` source set's manifest in Android Studio, and then it will only be included in debug builds.

Also, your device or emulator will need enough external storage to hold the file, which can get very large for long traces — 100MB a minute is well within reason.

Android Studio

On the per-thread details view of the Android Profiler's CPU graph, there is a red dot with a drop-down list to the left of it. This allows you to start method tracing from the IDE, rather than from Java code.

The drop-down offers two options:

- Sampled, where the data may miss some methods, but the analysis does not slow down the app
- Instrumented, where all method calls are collected, but the overhead may significantly slow down the app

After clicking the red dot to begin the method trace, it will turn into a gray square, and a timer will appear next to it to show how long you have been tracing. Click the gray square to stop method tracing.

Displaying Trace Data

If you collect the method trace data from within Android Studio, it is shown in the Android Profiler tool, beneath the CPU per-thread details view.

If you used `Debug.startMethodTracing()` and `Debug.stopMethodTracing()`, open the file from the Device File Explorer... at which point, nothing works, due to a bug in Android Studio 3.0. So, at least for the time being, you are limited to using the Android Profiler tool to collect your stack traces.

Interpreting Trace Data

Of course, the challenge is in making sense of what the IDE is trying to present. The UI that you get back, initially, is largely useless:

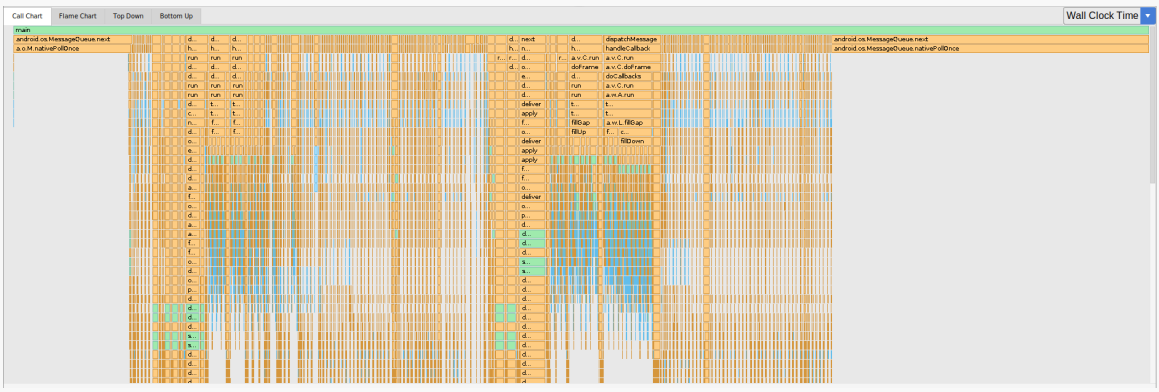


Figure 988: Method Tracing, Call Chart, As Initially Launched

This is the “call chart”, where each method gets a tiny box showing how much time it took compared to the other tiny boxes. Method calls are denoted by tiny boxes beneath a method’s tiny box.

The “flame chart” is more or less the same perspective, just upside-down:

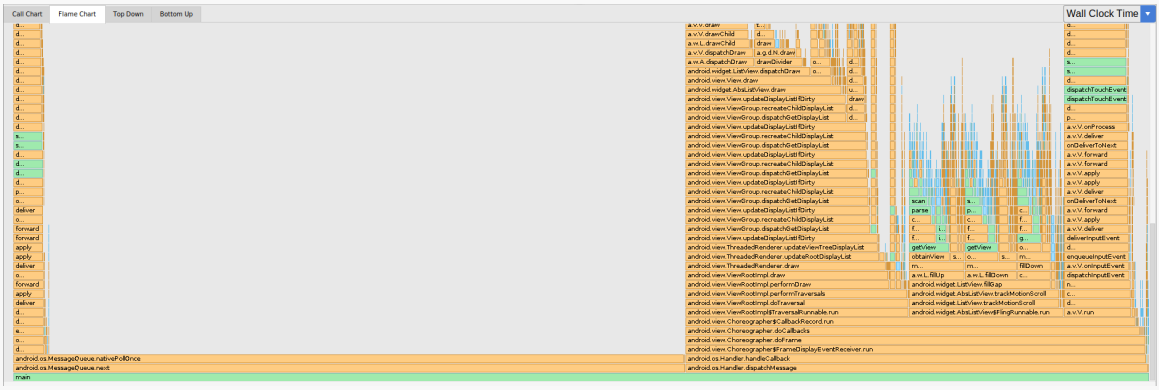


Figure 989: Method Tracing, Flame Chart, As Initially Launched

In both cases, your mouse scroll wheel can expand those tiny boxes to make them somewhat wider, but the font is extremely small, and there is no search facility to find anything in it.

Instead, switch over to the “Top Down” tab, which starts to become a bit more useful:

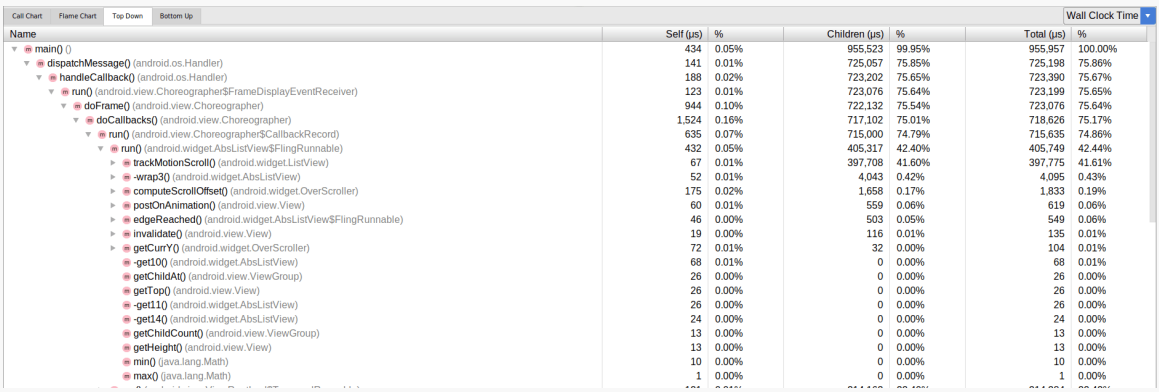


Figure 990: Method Tracing, "Top Down" Table, As Initially Launched

Here, you see the tree of method calls represented as a more classic tree, with an actually readable font.

The "Bottom Up" tab is even more useful, showing a simple table of all the distinct methods that were called:

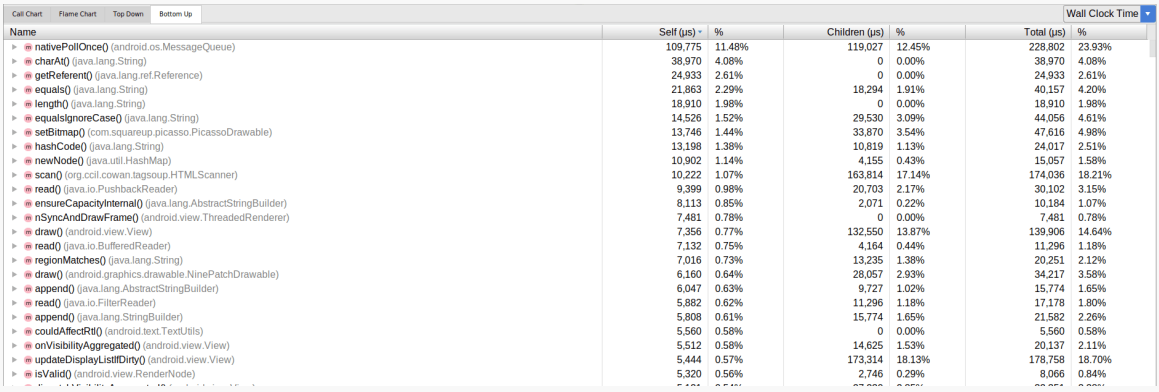


Figure 991: Method Tracing, "Bottom Up" Table, As Initially Launched

The two tables not only show methods and their classes, but also:

- How much time was taken directly in the method body for that method ("Self")
- How much time was taken by methods called by the method ("Children")
- How much total time was taken by the method, which is the sum of the first two columns? ("Total")

The values are shown in microseconds and as a percentage of the total.

What you are looking for are:

- References to your own classes and methods
- References to classes and methods from libraries that you are using
- Odd stuff that you do not recognize from previous examinations of this sort of method trace

If those sorts of things are taking significant time, as a percentage of the total, those are areas to investigate further:

- Are they being called too frequently?
- Are they intrinsically expensive?

Other General CPU Measurement Techniques

While method tracing can be useful for narrowing down a general performance issue to a specific portion of code, it does assume that you know approximately where the problem is, or that you even have a problem in the first place. There are other approaches to help you identify if and (roughly) where you have problems, which you can then attack with method tracing to try to refine.

Logging

Method tracing can be useful, if you have a rough idea of where your performance problem lies and need to narrow it down further. If you have a large and complicated application, though, trying to sift through all of it in method tracing may be difficult.

However, there is nothing stopping you from using good old-fashioned logging to get a rough idea of where your problems lie, for further analysis via method tracing. Just sprinkle your code with `Log.d()` calls, logging `SystemClock.uptimeMillis()` with an appropriate label to identify where you were at that moment in time. "Eyeballing" the Logcat output can illustrate areas where unexpected delays are occurring — the areas in which you can focus more time using method tracing.

A useful utility class for this is `TimingLogger`, in the `android.util` package. It will collect a series of "splits" and can dump them to Logcat along with the overall time between the creation of the `TimingLogger` object and the corresponding `dumpToLog()` method call. Note, though, that this will only log to Logcat when you call `dumpToLog()` — all of the calls to `split()` to record intermediate times have

their results buffered until `dumpToLog()` is called. Also note that logging needs to be set to VERBOSE for this information to actually be logged — use the command `adb shell setprop log.tag.LOG_TAG VERBOSE`, substituting your log tag (supplied to the `TimingLogger` constructor) for `LOG_TAG`.

FPS Calculations

Sometimes, it may not even be strictly obvious how bad the problem is. For example, consider scrolling a `ListView`. Some performance issues, like sporadic "hiccups" in the scrolling, will be visually apparent. However, absent those, it may be difficult to determine whether your particular `ListView` is behaving more slowly than you would expect.

A classic measurement for games is frames per second (FPS). Game developers aim for a high FPS value — 60 FPS is considered to be fairly smooth, for example. However, this sort of calculation can only really be done for applications that are continuously drawing – such as Romain Guy's `WindowBackground` sample application. Ordinary Android widget-based UIs are only drawing based upon user interaction or, possibly, upon background updates to data. In other words, if the UI will not even be trying to draw 60 times in a second, trying to measure FPS to get 60 FPS is pointless.

You may be able to achieve similar results, though, simply by logging how long it takes to, say, fling a list (use `setOnScrollListener()` and watch for `SCROLL_STATE_FLING` and other events).

UI "Jank" Measurement

A user interface is considered "janky" if it stutters or otherwise fails to operate smoothly, particularly during animated effects like scrolling. Sometimes, janky behavior is obvious to all. Sometimes, janky behavior is only noticeable to those sensitive to small hiccups in the UI.

This section will outline what "jank" is and how to determine, concretely, if your UI suffers from it.

What, Exactly, is Jank?

Prior to Android 4.0, it was difficult to come up with a concrete definition of jank. In effect, we were stuck with "I know it when I see it" ad-hoc analysis, rather than being able to rely on concrete measurements.

Project Butter changed that.

Android 4.0 ties all graphic operations to a 60 frames-per-second "vsync" frequency. If everything is working smoothly, your UI will update 60 times per second, uniformly (versus varying amounts of times between changes).

The converse is also true: if everything is not working smoothly, your UI will not update 60 times per second. This is the source of the term "dropped frames": when the time came around for an update, you were not ready, and that frame was skipped.

There are two main ways in which you will drop a frame:

1. You spend too much time on the main application thread, preventing Android from processing your requested UI updates in a timely fashion
2. Your UI changes are too complex to be rendered before time runs out for the current frame, causing your changes to spill over into the next frame

Each frame is ~16ms in duration on-screen (1/60th of a second). Hence, if we cause per-frame work to exceed 16ms, we will skip, or "drop", a frame.

So, what we need is some way to determine if our code is actually delivering frames on time.

Using gfxinfo

To determine if our problem is in the actual rendering of our UI updates, we can use the GPU profiling feature added in Android 4.2.

Enabling Developer Options

To toggle on GPU profiling, you will need to be able to get to the Developer Options portion of your Settings app. If you see this — typically towards the bottom of the list on the initial Settings screen — just tap on the entry.

If, however, Developer Options is missing, then you will need to use the super-secret trick for enabling Developer Options:

1. Tap on “About Phone”, “About Tablet”, or the equivalent at the bottom of your Settings list
2. Tap on the “Build Number” entry seven times in succession
3. Press BACK, and “Developer Options” should now be in the list

Toggling on GPU Profiling

There are two checkboxes in Developer Options that need to be checked for GPU profiling to be enabled.

The first is “Force GPU rendering”, in the Drawing section. As the name suggests, this will force your application to use the GPU for drawing, even if your application may have requested that hardware acceleration be disabled. Since most applications do not force hardware acceleration to be disabled, this checkbox probably will have no real effect on your app. Note that if you disabled hardware acceleration due to specific rendering problems, this checkbox will probably cause those rendering artifacts to re-appear during your testing.

The second is “Profile GPU rendering”, in the Monitoring section. This will cause the device to keep track of graphics performance on a per-process basis, in a way that we can dump later on.

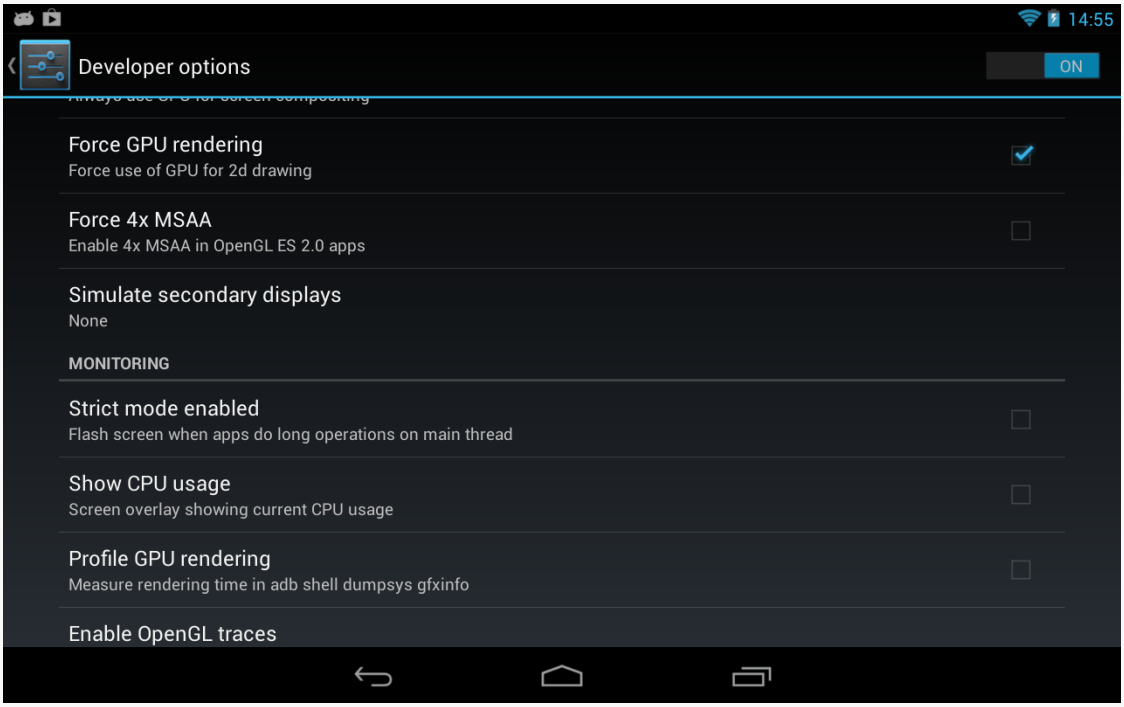


Figure 992: Developer Options, Showing "Force GPU rendering" and "Profile GPU rendering"

If your app was already running, you will need to get rid of its process (e.g., via swiping it off the recent-tasks list) after you check the "Profile GPU rendering" checkbox. At the present time, whether or not this profiling takes effect is determined at process startup time and is not changed on the fly when you toggle the checkbox. Besides, as noted above, starting with a fresh process should give you more accurate results.

Collecting Data

At this point, you can run your app and conduct your specific test, whether manually or via instrumentation (e.g., a targeted JUnit test suite).

When complete, run `adb shell dumpsys gfxinfo ...` in a terminal window, where `...` is replaced by the package name of your app (e.g., `com.commonsware.android.anim.threepane`). This will dump a fair amount of information to the terminal display:

```
mmurphy@xps15:~$ adb shell dumpsys gfxinfo com.commonsware.android.anim.threepane
Applications Graphics Acceleration Info:
```

```
Uptime: 482460 Realtime: 482454

** Graphics info for pid 3469 [com.commonsware.android.anim.threepane] **

Recent DisplayList operations
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    DrawPatch
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
```

```
                  Save
                  ClipRect
                  Translate
                  DrawText
                  RestoreToCount
     DrawPatch
   RestoreToCount

Caches:
Current memory usage / total memory usage (bytes):
  TextureCache          1078032 / 25165824
  LayerCache            7864320 / 16777216
  GradientCache               0 /   524288
  PathCache                   0 /  4194304
  CircleShapeCache            0 /  1048576
  OvalShapeCache              0 /  1048576
  RoundRectShapeCache         0 /  1048576
  RectShapeCache              0 /  1048576
  ArcShapeCache               0 /  1048576
  TextDropShadowCache         0 /  2097152
  FontRenderer 0         262144 /   262144
Other:
  FboCache                    3 /       16
  PatchCache                 89 /      512
Total memory usage:
  9204496 bytes, 8.78 MB

Profile data in ms:

  com.commonsware.android.anim.threepane/
com.commonsware.android.anim.threepane.MainActivity/android.view.ViewRootImpl@4131e788
  Draw  Process Execute
  14.45 59.67 10.44
  10.91 1.06  1.20
  1.73  12.80 1.19
  1.45  0.64  0.94
  2.15  0.47  0.57
  0.79  0.50  0.60
  2.23  0.49  0.73
  1.56  0.57  0.52
  6.14  0.47  1.92
  0.84  0.53  0.59
  1.58  0.52  0.60
  1.46  0.55  0.54
  1.74  0.75  0.68
  1.74  0.61  0.61
  1.05  0.62  1.00
  1.05  0.71  1.28
```

```
1.29  0.50  0.56
2.22  0.60  0.75
0.90  0.65  1.42
1.70  0.86  0.61
0.81  1.07  0.93
6.66  2.35  0.98
0.93  5.18  0.73
0.34  1.24  0.51
0.45  1.28  0.46
1.85  4.38  1.45
1.32  3.15  1.03
1.50  3.16  0.98
1.42  3.00  1.00
0.90  2.94  1.00
0.69  2.36  1.15
1.08  2.72  0.86
1.49  4.22  1.49
0.97  2.91  0.91
0.89  3.05  0.90
1.36  3.02  1.07
1.12  2.95  0.95
1.63  3.47  1.02
0.96  2.95  0.98
2.75  5.55  1.83
2.11  1.47  0.51
0.44  1.50  0.48
0.67  1.46  0.51
2.07  3.93  3.13
0.71  4.36  1.93
1.75  3.31  1.15
2.39  1.79  1.02
0.96  1.71  0.81
0.57  1.70  0.73
1.88  1.81  0.58
0.59  1.72  0.55
2.28  3.74  1.72
2.66  0.84  0.70
0.64  0.82  0.64
0.30  0.80  0.62
1.78  0.70  0.63
7.20  2.35  1.04
0.49  0.21  0.50
9.99  0.26  0.54
4.28  0.23  0.66
0.04  0.26  1.94
3.55  0.52  0.66
4.56  0.59  0.62
5.38  0.33  0.68
```

```
4.44  0.33  0.65
4.35  0.30  0.73
3.76  0.27  0.60
3.72  0.30  0.64
3.75  0.26  0.58
4.79  0.33  0.75
4.68  0.33  0.85
3.00  0.22  0.53
2.44  0.26  0.83
14.87 0.69  1.59
8.68  0.96  1.96
3.44  0.47  0.96
3.73  0.22  0.65
3.06  0.72  0.65
3.86  0.35  1.13
3.32  0.26  0.57
3.21  0.26  0.62
3.84  0.26  0.60
4.85  0.33  0.72
4.16  0.32  0.70
3.96  0.30  0.69
2.60  0.82  0.66
8.72  0.47  0.69
0.49  0.31  1.50
0.46  0.28  0.77
7.54  3.66  0.90
7.50  0.27  0.71
0.06  0.32  2.37
6.07  0.28  0.97
3.68  0.27  0.52
6.39  5.86  4.48
4.66  0.29  1.28
0.05  0.26  11.86
8.87  12.64 1.25
3.32  0.26  0.58
6.22  4.77  1.26
3.49  0.31  0.86
11.32 10.49 1.26
10.27 15.09 1.78
12.50 1.34  2.53
7.66  4.74  0.58
0.03  0.24  0.32
4.43  0.30  0.56
9.75  2.94  1.68
17.93 0.47  0.56
3.81  0.35  1.04
0.20  2.84  2.72
10.06 0.28  0.92
```

```
  5.74  0.72  1.92
  0.07  0.87  0.53
  2.05  0.95  2.03

View hierarchy:

  com.commonsware.android.anim.threepane/
com.commonsware.android.anim.threepane.MainActivity/android.view.ViewRootImpl@4131e788
  50 views, 4.48 kB of display lists, 115 frames rendered


Total ViewRootImpl: 1
Total Views:        50
Total DisplayList:  4.48 kB
```

We will discuss what this means in just a bit.

Disabling GPU Profiling

When you are done with your test, it is a good idea to undo the settings changes you made, at least "Profile GPU rendering". That way, the act of collecting this data does not itself add overhead to unrelated tests in the future.

Analyzing the Results

The key bit for our performance analysis is that long table labeled "Profile data in ms:". This reports, for a series of UI requests, how much time is spent:

- drawing your UI changes (e.g., `onDraw()` calls to various widgets and containers)
- processing the low-level drawing commands created via the draw phase, to create the contents of the frame
- executing the frame, sending it to the compositor to display on the screen

One way to interpret this table is to paste it into your favorite spreadsheet program, then use that program to draw a stacked column chart of the data. You can download a spreadsheet in ODS format (for use with LibreOffice, OpenOffice, or other tools that can handle that format) that contains the above table along with a stacked column chart:

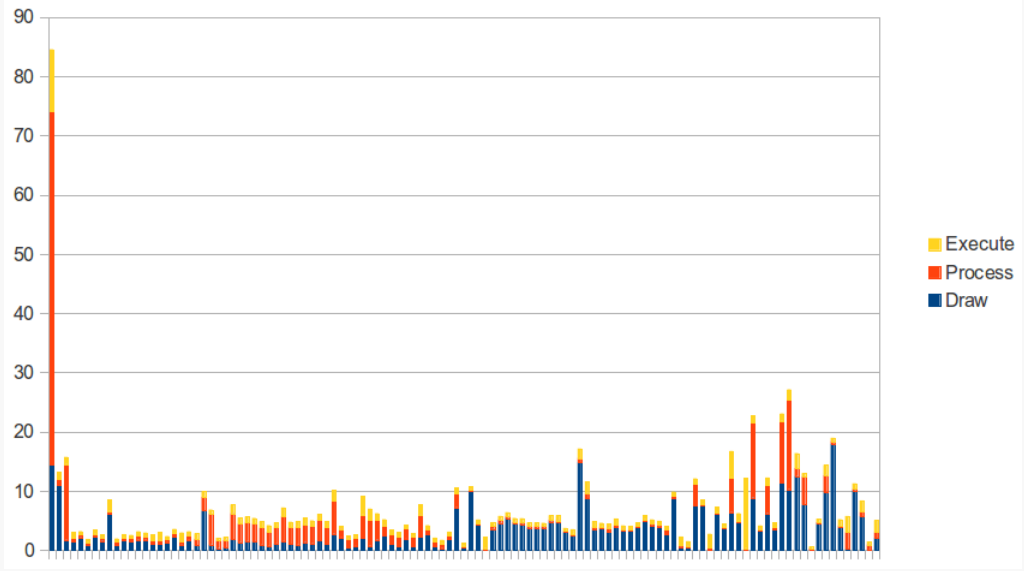


Figure 993: gfxinfo Output, In Stacked Column Chart

What you are looking for are columns that come close to, or exceed, the 16ms mark, with milliseconds on the Y axis. As you can see, many operations towards the end of the table are near or above 16ms, indicating that we are probably dropping some frames.

Using systrace

Another way we could determine whether or not we are dropping frames is to use `systrace` to collect system-level tracing information about the entire device, including our app.

`systrace` is a very powerful tool, one that 20 or 30 people on the planet truly understand, due to cryptic output and limited documentation. Using `gfxinfo` for detecting dropped frames is simple by comparison. On the other hand, `systrace` works for Android 4.1 and higher, versus the Android 4.2 requirement of `gfxinfo`.

Using `systrace` involves collecting a trace, which is saved in the form of an HTML file. The HTML file is then used to determine what went on during the period of the trace itself.

Enabling and Collecting a Trace: Command-Line

The original means of using `systrace` was from the command line. There is a `systrace.py` Python script located in the `tools/systrace/` directory of your SDK installation. If you have a Python interpreter (e.g., your development machine does not run Windows), you can use this approach.

To indicate what specific bits of information to collect, on Android 4.2 and higher, you can tap the "Enable traces" entry in the Monitoring section of the Developer Options page in Settings. This displays a multi-select dialog of the possible major categories of information that `systrace` should collect:

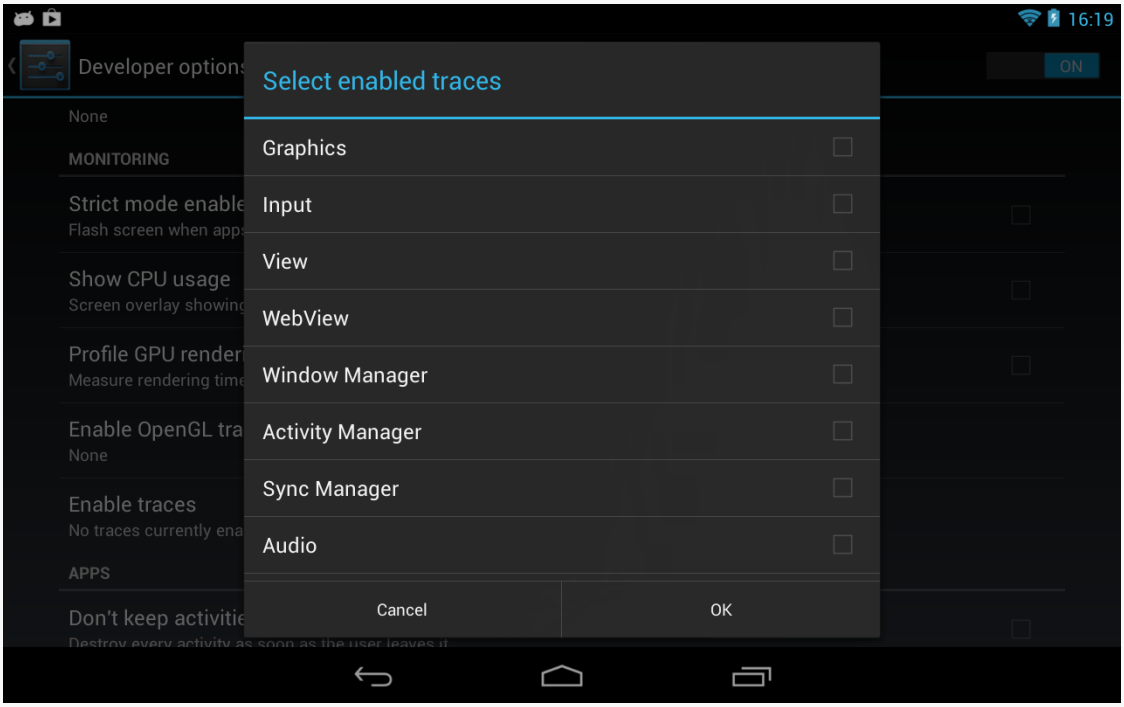


Figure 994: "Enable traces" In Settings

Alternatively, when you run the `systrace.py` script, you can include the `--set-tags` switch, with a comma-delimited list of specific traces ("tags") that you want to collect. The list of available tag names can be found in the developer documentation.

To actually collect the trace, you run the `systrace.py` script, optionally with `--set-tags` or other command-line switches.

On Android 4.1 and 4.2, this would look like:

```
python systrace.py --set-tags gfx,view,wm
adb shell stop
adb shell start
python systrace.py --time=10 -o trace.html
```

The first `python` command runs `systrace.py` just to set the tags to collect. If you set them using Developer Options in Settings, this would not be required. Restarting `adb shell` is apparently needed, for unclear reasons. The second `systrace.py` run will actually collect the trace, for 10 seconds (`--time=10`), resulting in report written to `trace.html` in the current working directory (`-o trace.html`).

The syntax changed for Android 4.3 and higher to simplify matters, combining the two `systrace.py` commands into one:

```
python systrace.py --time=10 -o trace.html gfx view sched wm
```

Note that `--set-tags` is no longer used. Instead, all values not identified by a switch are considered to be tags.

Once you run the script, quickly go to your device and run your test scenario, as the trace starts immediately upon running the script.

Enabling and Collecting a Trace: Android Device Monitor

The Android Device Monitor also allows you to collect a trace using `systrace`. There is a "Capture system wide trace using systrace" button in the toolbar in the Devices view, typically found in the DDMS perspective:

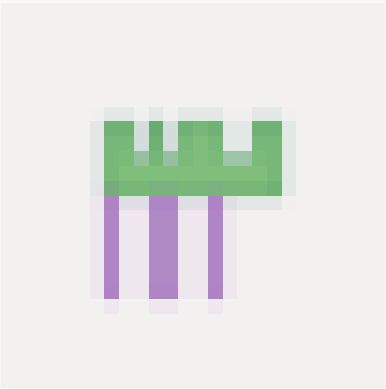


Figure 995: Systrace Toolbar Button in Devices View

To get to the Android Device Monitor in Android Studio, choose Tools > Android > Android Device Monitor from the main menu.

Tapping that toolbar button brings up a dialog that allows you to configure the trace you wish to collect, with checkboxes and fields replacing the variety of command-line switches you might use manually with `systrace.py`:

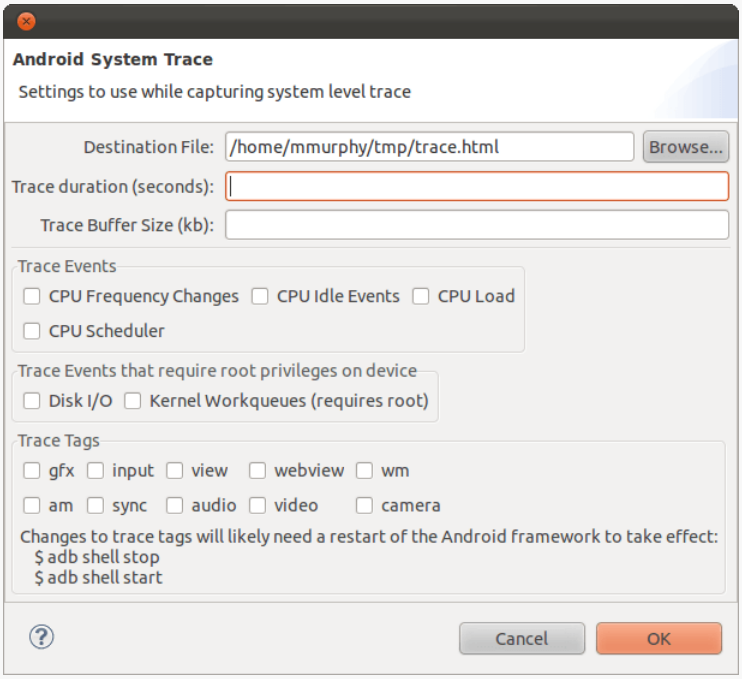


Figure 996: Systrace Dialog in Android Device Monitor

Notable settings that you will wish to tailor include:

- Where the trace will be written (by default, as `trace.html` in your home or user directory)
- The duration of the trace
- Which trace tags you wish to use

Clicking OK will then initiate the trace collection, at which point you will want to go to your test device and run through your test scenario.

Choosing the Trace Tags

All of these instructions have been telling you to specify what `systrace` tags to collect when you collect the trace data. So, what should you collect?

The big four are:

- `sched` for CPU scheduling
- `gfx` for graphics
- `view` for widget rendering
- `wm` for window management

Apps using `WebView` might consider the `webview` tag. There are a variety of other tags as well that you might find useful for one analysis or another. However, be careful not to request "everything but the kitchen sink", as it may make your reports difficult to interpret.

Also note that not all devices support all tags. `python systrace.py --list-categories` should tell you what is possible for your connected device.

Augmenting the Trace from Java

You can effectively add your own tag to the output in Java code, to flag key sections of application processing and see where they fall in the report's timeline. To do this:

- Add calls to `Trace.beginSection()` and `Trace.endSection()` in your API Level 18+ app. Here, `Trace` is `android.os.Trace`, and `beginSection()` takes a `String` parameter that you would like to have logged. Note that these calls can nest, so you can have one section inside of another, but `endSection()` closes the last-begun section. Hence, make sure that your `beginSection()` and `endSection()` calls match up, typically by using `try/finally` exception handling. Also, your `beginSection()` and `endSection()` calls must match up in terms of threads — you cannot begin a section on one thread and end it on another.
- Add the `--app` switch to name your application's package if you are running `systrace` from the command line. Or, in DDMS, choose your application in the "Enable Application Traces from" drop-down list.

Viewing and Interpreting the Results

What you get as output is an HTML file that can be viewed in the Chrome browser, though you will tend to want to use a development machine for this instead of, say, an Android tablet. That is because the navigation of the Web page is designed for use with a hardware QWERTY keyboard, which most Android devices lack.

You can find a sample trace from a Nexus 7 online, though note that the HTML is a bit large and may take a few seconds to download. Initially, you will see something like this:

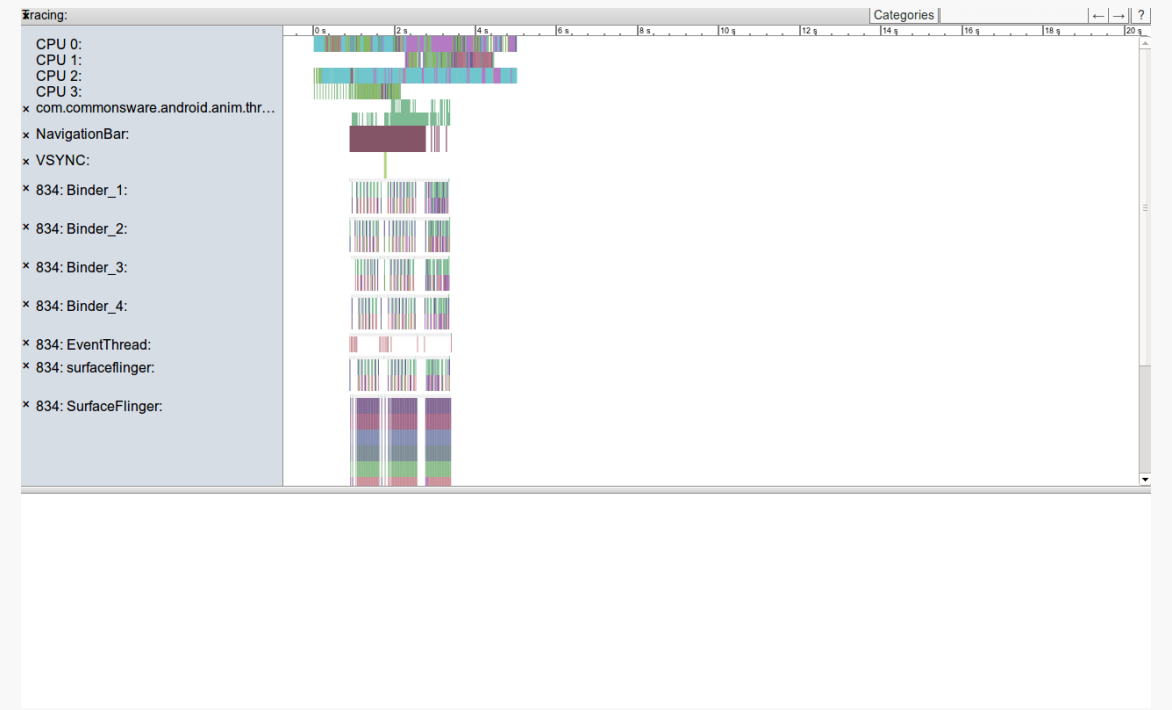


Figure 997: Systrace Output, As Initially Viewed

The left-hand sidebar represents various categories (or "slices" or "tags" or whatever) of data collected by `systrace`. The main area shows a timeline for the test, with rows corresponding to the sidebar entries for what was occurring at the various times for that particular category. The bottom pane will hold details that will appear when you click on various little blocks within that timeline.

Mostly, your navigation will use the W, A, S, and D keys, presumably chosen to make it appear as though you are playing a video game. Specifically:

- W will zoom in the timeline, while S will zoom out
- A and D will pane the timeline left and right

Jank will show up as gaps in the `SurfaceFlinger`:

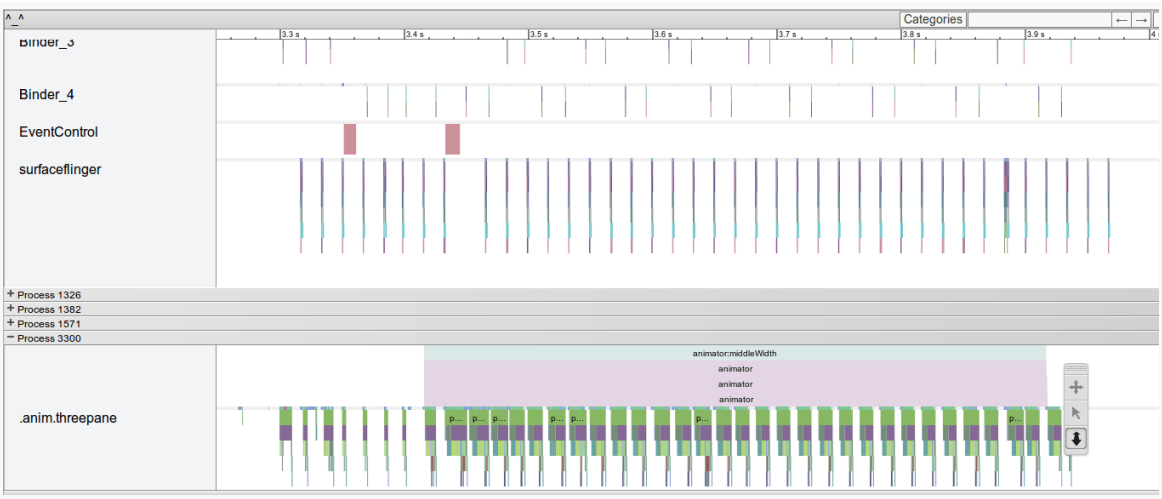


Figure 998: Systrace Output, Zoomed In on 0.7 Seconds of Profiling

Each of the major ticks across the timeline represents 0.1 seconds. There should be six frames in those seconds. However, we can see that in the 3.4-3.5 second range, there is a dropped frame, which shows up as a gap where there should be a pulse of `SurfaceFlinger` activity.

Zooming in further starts to bring up some detail for the threads in our process, showing methods within the view processing hierarchy that we were working on during this period of time:

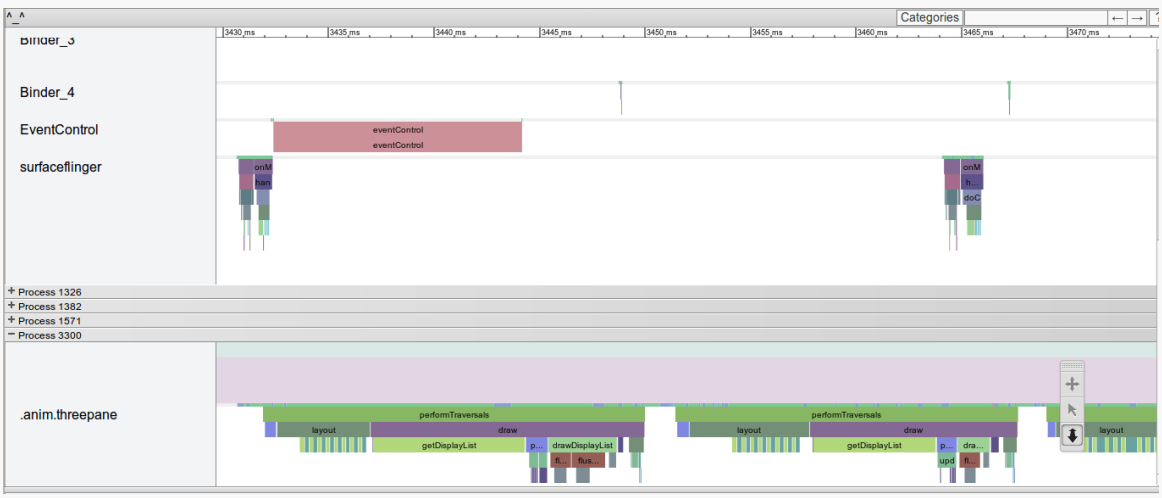


Figure 999: Systrace Output, Zoomed In on 40 Milliseconds of Profiling

In our `gfx/view` slice, we will see various blocks for different major operations in the rendering of our UI. Notably, you will see blocks labeled "performTraversals", referring to the private `performTraversals()` method on `ViewRootImpl`. It turns out that `performTraversals()` wraps around all of the work shown in the three columns of our `gfxinfo` output: draw, process, and execute. The widths of the "performTraversals" blocks in the `systrace` output shows us how long each of those takes. What we want are nice, short blocks. Instead, panning through our trace, you will see several that are too long. The chapter on "jank busting" will go into further analysis of where this particular sample application went wrong that caused this behavior.

Focus On: NDK

When Android was first released, many a developer wanted to run C/C++ code on it. There was little support for this, other than by distributing a binary executable and running it via a forked process. While this works, it is a bit cumbersome, and the process-based interface limits how cleanly your C/C++ code could interact with a Java-based UI. On top of all of that, the use of such binary executables is not well supported.

In June 2009, the core Android team released the Native Development Kit (NDK). This allows developers to write C/C++ for Android applications in a supported fashion, in the form of libraries linked to a hosting Java-based application via the Java Native Interface (JNI). This offers a wealth of opportunities for Android development, and this part of the book will explore how you can take advantage of the NDK to exploit those opportunities.

This chapter explains how to set up the NDK and apply it to your project. What it does not do is attempt to cover all possible uses of the NDK — game applications in particular have access to many frameworks, like OpenGL and OpenSL, that are beyond the scope of this book.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

This chapter also assumes that you know C/C++ programming.

The Role of the NDK

We start by examining Dalvik's primary limitation — speed. Next, we look at the reasons one might choose the NDK, speed among them. We wrap up with some reasons why the NDK may not be the right solution for every Android problem, despite its benefits.

Dalvik: Secure, Yes; Speedy, Not So Much

Dalvik was written with security as a high priority. Android's security architecture is built around Linux's user model, with each application getting its own user ID. With each application's process running under its own user ID, one process cannot readily affect other processes, helping to contain any single security flaw in an Android application or subsystem. This requires a fair number of processes. However, phones have limited RAM, and the Android project wanted to offer Java-based development. Multiple processes hosting their own Java virtual machines simply could not fit in a phone. Dalvik's virtual machine is designed to address this, maximizing the amount of the virtual machine that can be shared securely between processes (e.g., via "copy-on-write").

Of course, it is wonderful that Android has security so woven into the fabric of its implementation. However, inventing a new virtual machine required tradeoffs, and most of those are related to speed.

A fair amount of work has gone into making Java fast. Standard Java virtual machines do a remarkable job of optimizing applications on the fly, such that Java applications can perform at speeds near their C/C++ counterparts. This borders on the amazing and is a testament to the many engineers who put countless years into Java.

Dalvik, by comparison, is very young. Many of Java's performance optimization techniques — such as advanced garbage collection algorithms — simply have not been implemented to nearly the same level in Dalvik. This is not to say they will never exist, but it will take some time. Even then, though, there may be limits as to how fast Dalvik can operate, considering that it cannot "throw memory at the problem" to the extent Java can on the desktop or server.

ART has significantly improved matters, with ahead-of-time compilation (AOT) replacing just-in-time compilation (JIT) for getting native opcodes from the Dalvik

bytecodes. However, that code may still be inefficient when compared with writing C/C++ by hand.

Going Native

Java-based Android development via Dalvik and the Android SDK is far and away the option with the best support from the core Android team. HTML5 application development is another option that was brought to you by the core Android development team. The third leg of the official Android development triad is the NDK, provided to developers to address some specific problems, outlined below.

Speed

Far and away the biggest reason for using the NDK is speed, pure and simple. Writing in C/C++ for the device's CPU will be a major speed improvement over writing the same algorithms in Java, despite Android's JIT compiler (Dalvik) and AOT compiler (ART).

There is overhead in reaching out to the C/C++ code from a hosting Java application, and so for the best performance, you will want a coarse interface, without a lot of calls back and forth between Java and the native opcodes. This may require some redesign of what might otherwise be the "natural" way of writing the C/C++ code, or you may just have to settle for less of a speed improvement. Regardless, for many types of algorithms — from cryptography to game AI to video format conversions — using C/C++ with the NDK will make your application perform much better, to the point where it can enable applications to be successful that would be entirely too slow if written solely in Java.

Bear in mind, though, that much of what you think is Java code in your app really is native "under the covers". Many of the built-in Android classes are thin shims over native implementations. Again, focus on applying the NDK where you are performing lots of work yourself in Java code that might benefit from the performance gains.

Porting

You may already have some C/C++ code, written for another environment, that you would like to use with Android. That might be for a desktop application. That might be for another mobile platform, such as iOS, where C/C++ is an option. That might be for mobile platform, such as Symbian, where C/C++ is the conventional solution,

rather than some other language. Regardless, so long as that code is itself relatively platform-independent, it should be usable on Android.

This may significantly streamline your ability to support multiple platforms for your application, even if down-to-the-metal speed is not really something you necessarily need. This may also allow you to reuse existing C/C++ code written by others, for image processing or scripting languages or anything else.

Knowing Your Limits

Developers love silver bullets. Developers are forevermore seeking The One True Approach to development that will be problem-free. Sisyphus would approve, of course, as development always involves tradeoffs. So while the NDK's speed may make it tantalizing, it is not a solution for general Android application development, for several reasons, explored in this section.

Android APIs

The biggest issue with the NDK is that you have very limited access to Android itself. There are a few libraries bundled with Android that you can leverage, and a few other APIs offered specifically to the NDK, such as the ability to render OpenGL 3D graphics. But, generally speaking, the NDK has no access to the Android SDK, except by way of objects made available to it from the hosting application via JNI.

As such, it is best to view the NDK as a way of speeding up particular pieces of an SDK application — game physics, audio processing, OCR, and the like. All of those are algorithms that need to run on Android devices with data obtained from Android, but otherwise are independent of Android itself.

Cross-Platform Compatibility

While C/C++ *can* be written for cross-platform use, often it is not.

Sometimes, the disparity is one of APIs. Any time you use an API from a platform (e.g., iPhone) or a library (e.g., Qt) not available on Android, you introduce an incompatibility. This means that while a lot of your code — measured in terms of lines — may be fine for Android, there may be enough platform-specific bits woven throughout it that you would have a significant rewrite ahead of you to make it truly cross-platform.

Android itself, though, has a compatibility issue, in terms of CPUs. Android mostly runs on ARM devices today, since Android's initial focus was on smartphones, and ARM-powered smartphones at that. However, the focus on ARM will continue to waver, particularly as Android moves into other devices where other CPU architectures are more prevalent, such as Atom or MIPS for set-top boxes. While your code may be written in a fashion that works on all those architectures, the binaries that code produces will be specific to one architecture. The NDK gives you additional assistance in managing that, so that your application can simultaneously support multiple architectures.

Right now, the NDK supports ARM, x86, and MIPS CPU architectures. Of these, ARM CPUs power the *vast* majority of Android devices. The first generation of Google TV boxes, and a few other devices, use Intel x86 CPUs (usually Atom-based). MIPS is a relative newcomer to Android, with few devices using such CPUs at this time.

Introducing CWAC-AndDown

CWAC-AndDown is mentioned in passing in the chapter on rich text handling. It is an Android library that wraps hoedown, a C-based Markdown-to-HTML converter. The hoedown project itself is a fork of sundown, which itself was used by many sites, like GitHub, for their Markdown processing. CWAC-AndDown is great for projects that take in Markdown and want to render the results in a `WebView`.

Because CWAC-AndDown uses hoedown, and since hoedown is in C, CWAC-AndDown needs the NDK. We will examine how the NDK works using CWAC-AndDown as an example.

Installing the NDK

You can now use the NDK from Android Studio, which was not the case for quite some time.

In the Android Studio SDK Manager, you will need the CMake and NDK items from the SDK Tools tab:

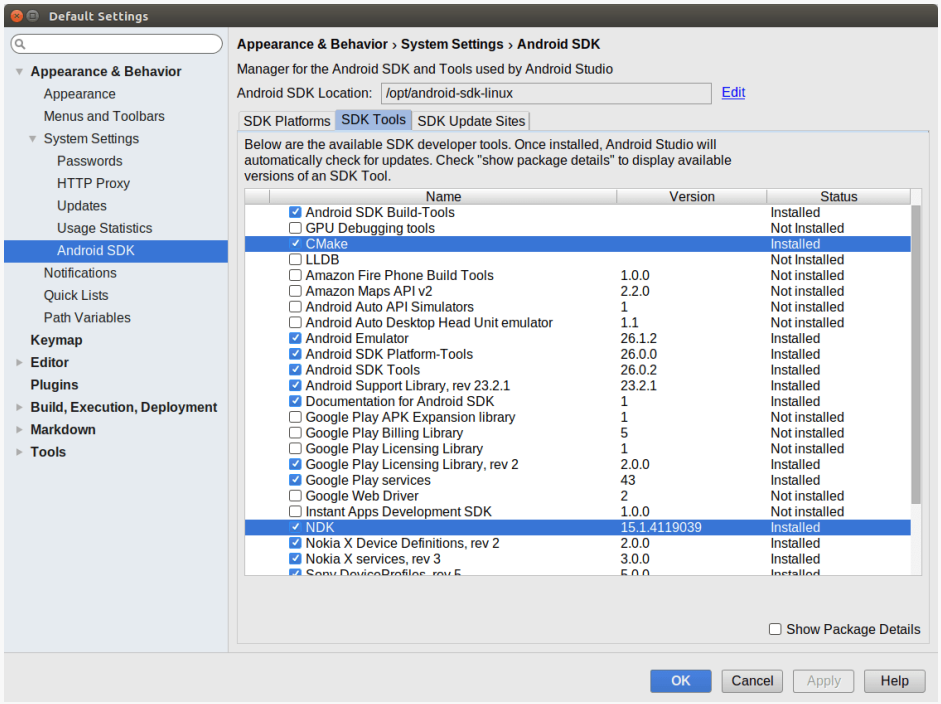


Figure 1000: NDK-Related Items from SDK Manager

Also note that when you create a new project, you will have the option of choosing to add C++ code support, which implies the NDK. This may cause the new-project wizard and later steps to install pieces of the NDK that you may be missing.

The Contents of an NDK Project

At its core, an NDK-enhanced Android project is a regular Android project. You still need a manifest, layouts, Java source code, and all the other trappings of a regular Android application. The NDK simply enables you to add C/C++ code to that project and have it included in your builds, referenced from your Java code via the Java Native Interface (JNI).

Your C/C++ Code

Android Studio tends to organize its code based on language. Hence, in your `main/` source set, in addition to `java/`, you can have a `cpp/` directory. Despite the name, both C and C++ code can reside in there, along with associated header files. These can be organized into whatever directory structure you want.

On the whole, your C/C++ code will be made up of two facets:

- The code doing the real work
- The code implementing your JNI interface

In the case of CWAC-AndDown, the hoedown source code is in a `hoedown/` directory off of the `cpp/` directory, while the JNI interface code is directly off of the `cpp/` directory:

Figure 1001: Android Studio Project Tree, Showing C Code

Your Makefile

The recommended build engine for the NDK in Android Studio is CMake, which is why that is an option available in the SDK Manager. CMake in Android Studio works off of a `CMakeLists.txt` file, which is a variation on the "makefile" sort of build instructions that has been used for C/C++ code for decades. This file goes in your module's directory (e.g., in `app/`, or in the case of CWAC-AndDown, in `anddown/`). It provides the details of what you want built in terms of native code:

```
# For more information about using CMake with Android Studio, read the
# documentation: https://d.android.com/studio/projects/add-native-code.html
```

```
# Sets the minimum version of CMake required to build the native library.

cmake_minimum_required(VERSION 3.4.1)

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
             anddown

             # Sets the library as a shared library.
             SHARED

             # Provides a relative path to your source file(s).
             src/main/cpp/anddown.c
             src/main/cpp/hoedown/autolink.c
             src/main/cpp/hoedown/buffer.c
             src/main/cpp/hoedown/document.c
             src/main/cpp/hoedown/escape.c
             src/main/cpp/hoedown/html.c
             src/main/cpp/hoedown/html_blocks.c
             src/main/cpp/hoedown/html_smartypants.c
             src/main/cpp/hoedown/stack.c
             src/main/cpp/hoedown/version.c)

include_directories(src/main/cpp src/main/cpp/hoedown)

set(CMAKE_CXX_FLAGS
   "${CMAKE_CXX_FLAGS} -std=c++0x -O2 -D_FORTIFY_SOURCE=2 -fstack-protector-all -fPIE")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -pie")
```

Here, we have four major configuration items.

- `cmake_minimum_required()` indicates the minimum required version of CMake that is likely to understand this `CMakeLists.txt` file and be able to compile your code
- `add_library()` indicates the name of the library (here, `anddown`), `SHARED` to indicate that we are building a shared library, then the paths to each of the individual C or C++ files that should be compiled into this library
- `include_directories()` provides the directories in which the C compiler (`gcc`) will be told to use to resolve any `#include` directives in the C/C++ code for pulling in headers
- `set()` sets environment variables used in the forked process for the C complier (in this case, setting some compile and link flags)

Presumably, CMake supports much more than this, but this should be sufficient for many simple uses of the NDK.

Your Gradle Changes

Your module's `build.gradle` file can have `externalNativeBuild` closures to configure CMake:

```
android {
    compileSdkVersion 26
    buildToolsVersion "26.0.2"

    defaultConfig {
        minSdkVersion 8
        targetSdkVersion 26

        testApplicationId "com.commonsware.cwac.anddown.test"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

            externalNativeBuild {
                cmake { cppFlags "" }
            }
        }
        debug {
            jniDebuggable true

            externalNativeBuild {
                cmake { cppFlags "-DDEBUG" }
            }
        }
    }

    externalNativeBuild {
        cmake { path "CMakeLists.txt" }
    }
}
```

Here, we:

- Tell CMake that our `CMakeLists.txt` file is in the designated file, for all build variants
- Tell CMake to make our `debug` builds debuggable, via the `-DDEBUG` flag and via the `jniDebuggable` Gradle setting

Building Your Library

Given all of the above, Android Studio will happily compile your NDK code as part of a build. Or, at the very least, it will try to. Unfortunately, at least as of Android

Studio 2.3.3, the integration between `gcc` and Android Studio is weak in terms of compile errors. If you have a syntax error in your native code — or, worse, have something wrong in `CMakeLists.txt` — you may have some challenges in understanding exactly what is wrong.

The compiled code will show up in `build/intermediates/cmake/` and, eventually, in the APK or AAR being assembled by this module. By default, Android Studio will build your native code for the following CPU architectures:

- `arm64-v8a`
- `armeabi`
- `armeabi-v7a`
- `mips`
- `mips64`
- `x86`
- `x86_64`

However, that means that you will wind up with a bigger APK than is necessary for any given user: they have an APK with N CPU architectures, installed on a device with only 1 CPU architecture. You can use "ABI splits" to arrange to also generate separate APKs for each CPU architecture. Some distribution channels, like the Play Store, support distributing separate APKs per CPU architecture for a single application ID.

Splits are configured using a `splits` closure inside the `android` closure:

```
android {
  splits {
    abi {
      enable true
      universalApk true
    }
  }
}
```

By default, ABI splits are disabled; the `enabled true` statement says that you want APKs split by CPU architecture. The `universalApk true` statement indicates that you want a "fat APK" supporting all CPU architectures in addition to a per-architecture APK. The "fat APK" would be useful for distribution channels that do not support per-architecture APKs.

By default, you get one APK for every CPU architecture. That is fine for cases where *you* are in control over the NDK code. If you are using a third-party library that has the NDK code, it may be that the library developers are supporting CPU architectures that you are not. In that case, you have two options:

1. Add an `exclude` statement to blacklist certain CPU architectures that you specifically do not want to support
2. Use `reset()` and `include` to whitelist certain CPU architectures that you specifically *do* want to support

```
android {
  splits {
    abi {
      enable true
      reset()
      include "x86", "armeabi-v7a", "x86", "x86_64", "arm64-v8a"
      universalApk true
    }
  }
}
```

Your Java and JNI Code

Now that you have your base C/C++ code being successfully compiled by the NDK, you need to turn your attention towards crafting the bridge between the Dalvik VM and the C/C++ code, following in the conventions of the Java Native Interface (JNI).

This section, while explaining the various steps involved in using the JNI, is far from a complete treatise on the subject. If you are going to spend a lot of time working with JNI, you are encouraged to seek additional resources on this topic, such as Core Java: Volume II, which has a chapter on JNI.

We want Android apps to be able to easily generate HTML from Markdown. To that end, we have an `AndDown` Java class containing two `public` methods:

- A one-parameter `markdownToHtml()`, which performs a basic conversion of Markdown to HTML, returning the `String` of HTML-formatted content
- A three-parameter `markdownToHtml()` method that also takes a pair of `int` bitfields for hoedown "extensions" and flags

The catch is that the three-parameter method has the `native` keyword, meaning that it is implemented in C/C++:

```
package com.commonsware.cwac.anddown;

/**
 * Java entry point to convert Markdown to HTML. Call the static markdownToHtml()
 * method for this.
 */
public class AndDown {
  static {
    System.loadLibrary("anddown");
  }

  /* block-level extensions */
  public static final int HOEDOWN_EXT_TABLES = (1 << 0);
  public static final int HOEDOWN_EXT_FENCED_CODE = (1 << 1);
  public static final int HOEDOWN_EXT_FOOTNOTES = (1 << 2);

  /* span-level extensions */
  public static final int HOEDOWN_EXT_AUTOLINK = (1 << 3);
  public static final int HOEDOWN_EXT_STRIKETHROUGH = (1 << 4);
  public static final int HOEDOWN_EXT_UNDERLINE = (1 << 5);
  public static final int HOEDOWN_EXT_HIGHLIGHT = (1 << 6);
  public static final int HOEDOWN_EXT_QUOTE = (1 << 7);
  public static final int HOEDOWN_EXT_SUPERSCRIPT = (1 << 8);
  public static final int HOEDOWN_EXT_MATH = (1 << 9);

  /* other flags */
  public static final int HOEDOWN_EXT_NO_INTRA_EMPHASIS = (1 << 11);
  public static final int HOEDOWN_EXT_SPACE_HEADERS = (1 << 12);
  public static final int HOEDOWN_EXT_MATH_EXPLICIT = (1 << 13);

  /* negative flags */
  public static final int HOEDOWN_EXT_DISABLE_INDENTED_CODE = (1 << 14);

  public static final int HOEDOWN_HTML_SKIP_HTML = (1 << 0);
  public static final int HOEDOWN_HTML_ESCAPE = (1 << 1);
  public static final int HOEDOWN_HTML_HARD_WRAP = (1 << 2);
  public static final int HOEDOWN_HTML_USE_XHTML = (1 << 3);

  /**
   * Given Markdown, returns HTML. 'Nuff said.
   *
   * @param raw Markdown-formatted string
   * @return HTML-formatted string
   */
  public String markdownToHtml(String raw) {
    return(markdownToHtml(raw, 0, 0));
  }

  public native String markdownToHtml(String raw, int extensions, int flags);
}
```

AndDown also has a `static` block with a `System.loadLibrary()` call, which teaches the virtual machine what native library to load and look for the `native` methods. In this case, the library is anddown, which is the name that we gave it in `CMakeLists.txt`.

The Java JDK comes with a `javah` utility. Given a Java class like `AndDown`, `javah` will generate the appropriate C header file for all `native` methods. That gives us the awkwardly-named `com_commonsware_cwac_anddown_AndDown.h` file, with the equally-awkward `Java_com_commonsware_cwac_anddown_AndDown_markdownToHtml()` function declaration:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_commonsware_cwac_anddown_AndDown */

#ifndef _Included_com_commonsware_cwac_anddown_AndDown
#define _Included_com_commonsware_cwac_anddown_AndDown
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_commonsware_cwac_anddown_AndDown
 * Method:    markdown
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_commonsware_cwac_anddown_AndDown_markdownToHtml
  (JNIEnv *, jobject, jstring, unsigned int, unsigned int);

#ifdef __cplusplus
}
#endif
#endif
```

We do not really use this file much, other than to get the C function prototype that we need to implement. The JNI bridge code will look for that function when our Java code calls the `native` implementation of `markdownToHtml()`. Our implementation of that C function delegates to a series of hoedown functions for actually performing the Markdown-to-HTML conversion:

```
/***
  Copyright (c) 2010 CommonsWare, LLC
  Portions (c) somebody else who didn't bother to indicate who they were

  Licensed under the Apache License, Version 2.0 (the "License"); you may
  not use this file except in compliance with the License. You may obtain
  a copy of the License at
    http://www.apache.org/licenses/LICENSE-2.0
  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

#include "com_commonsware_cwac_anddown_AndDown.h"
#include "document.h"
#include "html.h"
#include "buffer.h"
```

```
#define INPUT_UNIT 64
#define OUTPUT_UNIT 64

JNIEXPORT jstring JNICALL Java_com_commonsware_cwac_anddown_AndDown_markdownToHtml
  (JNIEnv *env, jobject o, jstring raw, unsigned int extensions, unsigned int flags) {
  struct hoedown_buffer *ib, *ob;
  jstring result;
  hoedown_renderer *renderer;
  hoedown_document *document;
  const char* str;

  str = (*env)->GetStringUTFChars(env, raw, NULL);

  ib = hoedown_buffer_new(INPUT_UNIT);
  hoedown_buffer_puts(ib, str);
  ob = hoedown_buffer_new(OUTPUT_UNIT);

  (*env)->ReleaseStringUTFChars(env, raw, str);

  renderer = hoedown_html_renderer_new(flags, 0);
  document = hoedown_document_new(renderer, extensions, 16);

  hoedown_document_render(document, ob, ib->data, ib->size);
  hoedown_document_free(document);

  result=(*env)->NewStringUTF(env, hoedown_buffer_cstr(ob));

  /* cleanup */
  hoedown_buffer_free(ib);
  hoedown_buffer_free(ob);

  return(result);
}
```

However, our Java code is largely oblivious to this. It can simply call the `markdownToHtml()` methods on `AndDown`, such as in a trivial instrumentation test case for confirming that the build works:

```
package com.commonsware.cwac.anddown.test;

import android.support.test.runner.AndroidJUnit4;
import com.commonsware.cwac.anddown.AndDown;
import junit.framework.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class MarkdownTest {
  @Test
  public void testSimple() {
    AndDown andDown=new AndDown();

    Assert.assertEquals("<p>Hello, world</p>",
```

```
      andDown.markdownToHtml("Hello, world").trim());
    Assert.assertEquals("<p>This <q>contains</q> a quote</p>",
      andDown.markdownToHtml("This \"contains\" a quote",
        AndDown.HOEDOWN_EXT_QUOTE, 0).trim());
  }
}
```

libhoudini and the NDK

`libhoudini` is a proprietary ARM translation layer for x86-powered Android devices. It allows an app that has NDK binaries for ARM, but not x86, to still run on x86 hardware, albeit not as quickly as it would with native x86 binaries. The advantage is that you can ship an APK without x86 binaries (so the APK is smaller) yet still be able to run on x86 devices.

Given ARM's current dominance in the Android ecosystem, `libhoudini` is hugely useful for Intel and hardware vendors interested in using Intel's mobile CPUs. Without it, only apps that ship x86 NDK binaries would be compatible with x86-powered devices like the Samsung Galaxy Tab 3 10.1" tablet. Some developers probably skip x86 NDK binaries, because they are not aware of popular x86-powered devices, or lack one for testing, or are concerned over APK size. The Play Store for x86 would shrink substantially from the million-plus apps available to ARM devices, to those that do not use the NDK or happen to ship x86 binaries. `libhoudini` makes ARM-only NDK binaries usable on x86, giving x86-powered Android devices access to more of the Play Store catalog.

However, it *is* slower. A test suite for SQLCipher for Android, run on an ASUS MeMO Pad FHD 10, ran about three times as long when using the ARM binaries and `libhoudini`, when compared to the same test run using x86 binaries. On the other hand, supporting x86 in addition to ARM adds another 5MB to the app, on top of the 6.5MB spent for ARM and the platform-neutral pieces. Being able to use SQLCipher for Android without the x86 binaries might be useful, particularly for apps bumping up against APK size limits, like the 50MB limit on the Play Store.

You may wish to do your own testing. Testing is easy enough: just temporarily move the `x86/` directory from `libs/` somewhere else, then recompile and test on `libhoudini`-equipped hardware. If you do not have your own `libhoudini`-equipped hardware, you may be able to take advantage of services like Samsung's Remote Test Lab, which recently added the Galaxy Tab 3 10.1 to its lineup.

If you have the space for it, include the x86 binaries for your NDK-compiled libraries. This will give you maximum speed for little incremental engineering cost. However, if space is at a premium, `libhoudini` may allow you to reach many of the same x86 devices, but be sure that your app will run acceptably given the performance overhead.

Improving CPU Performance in Java

Knowing that you have CPU-related issues in your app is one thing — doing something about it is the next challenge. In some respects, tuning an Android application is a "one-off" job, tied to the particulars of the application and what it is trying to accomplish. That being said, this chapter will outline some general-purpose ways of boosting performance that may counter issues that you are running into.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

Reduce CPU Utilization

One class of CPU-related problems come from purely sluggish code. These are the sorts of things you will see in Traceview, for example – methods or branches of code that seem to take an inordinately long time. These are also some of the most difficult to have general solutions for, as often times it comes down to what the application is trying to accomplish. However, the following sections provide suggestions for consuming fewer CPU instructions while getting the same work done.

These are presented in no particular order.

Standard Java Optimizations

Most of your algorithm fixes will be standard Java optimizations, no different than have been used by Java projects over the past decade and change. This section outlines a few of them. For more, consider reading *Effective Java* by Joshua Bloch or *Java Performance Tuning* by Jack Shirazi.

Avoid Excessive Synchronization

Few objects in `java.*` namespaces are intrinsically thread-safe, outside of `java.util.concurrent`. Typically, you need to perform your own synchronization if multiple threads will be accessing non-thread-safe objects. However, sometimes, Java classes have synchronization that you neither expect nor need. Synchronization adds unnecessary overhead.

The classic example here is `StringBuffer` and `StringBuilder`. `StringBuffer` was part of Java from early on, and, for whatever reason, was written to be thread-safe — two threads that append to the buffer will not cause any problems. However, most of the time, you are only using the `StringBuffer` from one thread, meaning all that synchronization overhead is a waste. Later on, Java added `StringBuilder`, with the same basic set of methods as has `StringBuffer`, but without the synchronization.

Similarly, in your own code, only synchronize where it is really needed. Do not toss the `synchronized` keyword around randomly, or use concurrent collections that will only be used by one thread, etc.

Avoid Floating-Point Math

The first generation of Android devices lacked a floating-point coprocessor on the ARM CPU package. As a result, floating-point math speed was atrocious. That is why the Google Maps add-on for Android uses `GeoPoint`, with latitude and longitude in integer microdegrees, rather than the standard Android `Location` class, which uses Java double variables holding decimal degrees.

While later Android devices do have floating-point coprocessor support, that does not mean that floating-point math is now as fast as integer math. If you find that your code is spending lots of time on floating-point calculations, consider whether a change in units would allow you to replace the floating-point calculations with integer equivalents. For example, microdegrees for latitude and longitude provide

adequate granularity for most maps, yet allow Google Maps to do all of its calculations in integers.

Similarly, consider whether the full decimal accuracy of floating-point values is really needed. While it may be physically possible to perform distance calculations in meters with accuracy to a few decimal points, for example, in many cases the user will not need that degree of accuracy. If so, perhaps changing to fixed-point (integer) math can boost your performance.

Don't Assume Built-In Algorithms are Best

Years upon years of work has gone into the implementation of various algorithms that underlie Java methods, like searching for substrings inside of strings.

Somewhat less work has gone into the implementation of the Apache Harmony versions of those methods, simply because the project is younger, and it is a modified version of the Harmony implementation that you will find in Android. While the core Android team has made many improvements to the original Harmony implementation, those improvements may be for optimizations that do not fit your needs (e.g., optimizing to reduce memory consumption at the expense of CPU time).

But beyond that, there are dozens of string-matching algorithms, some of which may be better for you depending on the string being searched and the string being searched for. Hence, you may wish to consider applying your own searching algorithm rather than relying on the built-in one, to boost performance. And, this same concept may hold for other algorithms as well (e.g., sorting).

Of course, this will also increase the complexity of your application, with long-term impacts in terms of maintenance cost. Hence, do not assume the built-in algorithms are the worst, either — optimize those algorithms that Traceview or logging suggest are where you are spending too much time.

Support Hardware-Accelerated Graphics

An easy "win" is to add `android:hardwareAccelerated="true"` to your `<application>` element in the manifest. This toggles on hardware acceleration for 2D graphics, including much of the stock widget framework. For maximum backwards compatibility, this hardware acceleration is off, but adding the aforementioned attribute will enable it for all activities in your application.

Note that this is only available starting with Android 3.0. It is safe to have the attribute in the manifest for older Android devices, as they simply will ignore your request.

You also should test your application thoroughly after enabling hardware acceleration, to make sure there are no unexpected issues. For ordinary widget-based applications, you should encounter no problems. Games or other applications that do their own drawing might have issues. If you find that some of your code runs into problems, you can override hardware acceleration on a per-activity basis by putting the `android:hardwareAccelerated` on `<activity>` elements in the manifest.

Minimize IPC

Calling a method on an object in your own process is fairly inexpensive. The overhead of the method invocation is fairly minuscule, and so the time involved is simply however long it takes for that method to do its work.

Invoking behaviors in another process, via inter-process communication (IPC), is considerably more expensive. Your request has to be converted into a byte array (e.g., via the `Parcelable` interface), made available to the other process, converted back into a regular request, then executed. This adds substantial CPU overhead.

There are three basic flavors of IPC in Android:

1. "Directly" invoking a third-party application's service's AIDL-published interface, to which you bound with `bindService()`
2. Performing operations on a content provider that is not part of your application (i.e., supplied by the OS or a third-party application)
3. Performing other operations that, under the covers, trigger IPC

Remote Bound Service

Using a remote service is fairly obvious when you do it — it is difficult to mistake copying the AIDL into your project and such. The proxy object generated from the AIDL converts all your method calls on the interface into IPC operations, and this is relatively expensive.

If you are exposing a service via AIDL, design your API to be coarse-grained. Do not require the client to make 1,000 method invocations to accomplish something that can be done in 1 via slightly more complex arguments and return values.

If you are consuming a remote service, try not to get into situations where you have to make lots of calls in a tight loop, or per row of a scrolled `AdapterView`, or anything else where the overhead may become troublesome.

For example, in the `CPU-Java/AIDLOverhead` sample project, you will find a pair of projects implementing the same do-nothing method in equivalent services. One uses AIDL and is bound to remotely from a separate client application; the other is a local service in the client application itself. The client then calls the do-nothing method 1 million times for each of the two services. On average, on a Samsung Galaxy Tab 10.1, 1 million calls takes around 170 seconds for the remote service, while it takes around 170 *milliseconds* for the local service. Hence, the overhead of an individual remote method invocation is small (~170 microseconds), but doing lots of them in a loop, or as the user flings a `ListView`, might become noticeable.

Remote Content Provider

Using a content provider can be somewhat less obvious of a problem. Using `ContentResolver` or a `CursorLoader` looks the same whether it is your own content provider or someone else's. However, you know what content providers you wrote; anything else is probably running in another process.

As with remote services, try to aggregate operations with remote content providers, such as:

1. Use `bulkInsert()` rather than lots of individual `insert()` calls
2. Try to avoid calling `update()` or `delete()` in a tight loop – instead, if the content provider supports it, use a more complex "WHERE clause" to update or delete everything at once
3. Try to get all your data back in few queries, rather than lots of little ones... though this can then cause you issues in terms of memory consumption

Remote OS Operation

The content provider scenario is really a subset of the broader case where you request that Android do something for you and winds up performing IPC as part of that.

Sometimes, this is going to be obvious. If you are sending commands to a third-party service via `startService()`, by definition, this will involve IPC, since the third-party

service will run in a third-party process. Try to avoid calling `startService()` lots of times in close succession.

However, there are plenty of cases that are less obvious:

1. All requests to `startActivity()`, `startService()`, and `sendBroadcast()` involve IPC, as it is a separate OS process that does the real work
2. Registering and unregistering a `BroadcastReceiver` (e.g., `registerReceiver()`) involves IPC
3. All of the "system services", such as `LocationManager`, are really rich interfaces to an AIDL-defined remote service, and so most operations on these system services require IPC

Once again, your objective should be to minimize calls that involve IPC, particularly where you are making those calls frequently in close succession, such as in a loop. For example, frequently calling `getLastKnownLocation()` will be expensive, as that involves IPC to a system process.

Android-Specific Java Optimizations

The way that the Dalvik VM was implemented and operates is subtly different than a traditional Java VM. Therefore, there are some optimizations that are more important on Android than you might find in regular desktop or server Java.

The Android developer documentation has a roster of such optimizations. Some of the highlights include:

1. Getters and setters, while perhaps useful for encapsulation, are significantly slower than direct field access. For simpler cases, such as `ViewHolder` objects for optimizing an Adapter, consider skipping the accessor methods and just use the fields directly.
2. Some popular method calls are replaced by hand-created assembler instructions rather than code generated via the JIT compiler. `indexOf()` on `String` and `arraycopy()` on `System` are two cited examples. These will run much faster than anything you might create yourself in Java.

Reduce Time on the Main Application Thread

Another class of CPU-related problem is when your code may be efficient, but it is occurring on the main application thread, causing your UI to react sluggishly. You

might have tuned your decryption algorithm as best as is mathematically possible, but it may be that decrypting data on the main application thread simply takes too much time. Or, perhaps `StrictMode` complained about some disk or network I/O that you are performing on the main application thread.

The following sections recap some commonly-seen patterns for moving work off the main application thread, plus a few newer options that you may have missed.

Generate Less Garbage

Most developers think of having too many allocations as being solely an issue of heap space. That certainly has an impact, and depending on the nature of the allocations (e.g., bitmaps), it may be the dominant issue.

However, garbage has impacts from a CPU standpoint as well. Every object you create causes its constructor to be executed. Every object that is garbage-collected requires CPU time both to find the object in the heap and to actually clean it up (e.g., execute the finalizer, if any).

Worse still, on older versions of Android (e.g., Android 2.2 and down), the garbage collector interrupts the entire process to do its work, so the more garbage you generate, the more times you "stop the world". Game developers have had to deal with this since Android's inception. To maintain a 60 FPS refresh rate, you cannot afford *any* garbage collections on older devices, as a single GC run could easily take more than the ~16ms you have per drawing pass.

As a result of all of this, game developers have had to carefully manage their own object pools, pre-allocating a bunch of objects before game play begins, then using and recycling those objects themselves, only allowing them to become garbage after game play ends.

Most non-game Android applications may not have to go to quite that extreme across the board. However, there are cases where excessive allocation may cause you difficulty. For example, avoiding creating too much garbage is one aspect of view recycling with `AdapterView`, which is covered in greater detail in the next section.

If Traceview indicates that you are spending a lot of time in garbage collection, pay attention to your loops or things that may be invoked many times in rapid succession (e.g., accessing data from a custom `Cursor` implementation that is tied to a `CursorAdapter`). These are the most likely places where your own code might be creating lots of extra objects that are not needed. Examining the heap to see what is

all being created (and eventually garbage collected) will be covered in an upcoming chapter of the book.

View Recycling

Perhaps the best-covered Android-specific optimization is view recycling with `AdapterView`.

In a nutshell, if you are extending `BaseAdapter`, or if you are overriding `getView()` in another adapter, please make use of the `View` parameter supplied to `getView()` (referred to here as `convertView`). If `convertView` is not `null`, it is one of your previous `View` objects you returned from `getView()` before, being offered to you for recycling purposes. Using `convertView` saves you from inflating or manually constructing a fresh `View` every time the user scrolls, and both of those operations are relatively expensive.

If you have been ignoring `convertView` because you have more than one type of `View` that `getView()` returns, your `Adapter` should be overriding `getViewTypeCount()` and `getItemViewType()`. These will allow Android to maintain separate object pools for each type of row from your `Adapter`, so `getView()` is guaranteed to be passed a `convertView` that matches the row type you are trying to create.

A somewhat more advanced optimization — caching all those `findViewById()` lookups — is also possible once your row recycling is in place. Often referred to as "the holder pattern", you do the `findViewById()` calls when you inflate a new row, then attach the `findViewById()` results to the row itself via some custom "holder" object and the `setTag()` method on `View`. When you recycle the row, you can get your "holder" back via `getTag()` and skip having to do the `findViewById()` calls again.

Background Threads

Of course, the backbone of any strategy to move work off the main application thread is to use background threads, in one form or fashion. You will want to apply these in places where `StrictMode` complains about network or disk I/O, or places where Traceview or logging indicate that you are taking too much time on the main application thread during GUI processing (e.g., converting downloaded bitmap images into `Bitmap` objects via `BitmapFactory`).

Sometimes, you will manually dictate where work should be done in the background, either by forking threads yourself or by using `AsyncTask`. `AsyncTask` is a

nice framework, handling all of the inter-thread communication for you and neatly packaging up the work to be done in readily understood methods. However, `AsyncTask` does not fit every scenario — it is mostly designed for "transactional" work that is known to take a modest amount of time (milliseconds to seconds) then end. For cases where you need unbounded background processing, such as monitoring a socket for incoming data, forking your own thread will be the better approach.

Sometimes, you will use facilities supplied by Android to move work to the background. For example, many activities are backed by a `Cursor` obtained from a database or content provider. Classically, you would manage the cursor (via `startManagingCursor()`) or otherwise arrange to refresh that `Cursor` in `onResume()`, so when your activity returns to the foreground after having been gone for a while, you would have fresh data. However, this pattern tends to lead to database I/O on the main application thread, triggering complaints from `StrictMode`. Android 3.0 and the Android Compatibility Library offer a `Loader` framework designed to try to solve the core pattern of refreshing the data, while arranging for the work to be done asynchronously.

Asynchronous BroadcastReceiver Operations

99.44% of the time (approximately) that Android calls your code in some sort of event handler, you are being called on the main application thread. This includes manifest-registered `BroadcastReceiver` components — `onReceive()` is called on the main application thread. So any work you do in `onReceive()` ties up that thread (possibly impacting an activity of yours in the foreground), and if you take more than 10 seconds, Android will terminate your `BroadcastReceiver` with extreme prejudice.

Classically, manifest-registered `BroadcastReceiver` components only live as long as the `onReceive()` call does, meaning you can do very little work in the `BroadcastReceiver` itself. The typical pattern is to have it send a command to a service via `startService()`, where the service "does the heavy lifting".

Android 3.0 added a `goAsync()` method on `BroadcastReceiver` that can help a bit here. While under-documented, it tells Android that you need more time to complete the broadcast work, but that you can do that work on a background thread. This does not eliminate the 10-second rule, but it does mean that the `BroadcastReceiver` can do some amount of I/O without having to send a command to a service to do it while still not tying up the main application thread.

The CPU-Java/GoAsync sample project demonstrates `goAsync()` in use, as the project name might suggest.

Our activity's layout consists of two `Button` widgets and an `EditText` widget:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="match_parent"
  android:layout_height="match_parent">
  <EditText android:id="@+id/editText1" android:layout_width="match_parent"
    android:layout_height="wrap_content">
  </EditText>
  <Button android:layout_width="match_parent" android:id="@+id/button1"
    android:layout_height="wrap_content" android:text="@string/nonasync"
    android:onClick="sendNonAsync"></Button>
  <Button android:layout_width="match_parent" android:id="@+id/button2"
    android:layout_height="wrap_content" android:text="@string/async"
    android:onClick="sendAsync"></Button>
</LinearLayout>
```

(from CPU-Java/GoAsync/app/src/main/res/layout/main.xml)

The activity itself simply has `sendAsync()` and `sendNonAsync()` methods, each invoking `sendBroadcast()` to a different `BroadcastReceiver` implementation:

```
package com.commonsware.android.tuning.goasync;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class GoAsyncActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void sendAsync(View v) {
    sendBroadcast(new Intent(this, AsyncReceiver.class));
  }

  public void sendNonAsync(View v) {
    sendBroadcast(new Intent(this, NonAsyncReceiver.class));
  }
}
```

(from CPU-Java/GoAsync/app/src/main/java/com/commonsware/android/tuning/goasync/GoAsyncActivity.java)

The `NonAsyncReceiver` simulates doing time-consuming work in `onReceive()` itself:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class NonAsyncReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context arg0, Intent arg1) {
    SystemClock.sleep(7000);
  }
}
```

(from CPU-Java/GoAsync/app/src/main/java/com/commonsware/android/tuning/goasync/NonAsyncReceiver.java)

Hence, if you click the "Send Non-Async Broadcast" button, not only will the button fail to return to its normal state for seven seconds, but the `EditText` will not respond to user input either.

The `AsyncReceiver`, though, uses `goAsync()`:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class AsyncReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    final BroadcastReceiver.PendingResult result=goAsync();

    (new Thread() {
      public void run() {
        SystemClock.sleep(7000);
        result.finish();
      }
    }).start();
  }
}
```

(from CPU-Java/GoAsync/app/src/main/java/com/commonsware/android/tuning/goasync/AsyncReceiver.java)

The `goAsync()` method returns a `PendingResult`, which supports a series of methods that you might ordinarily fire on the `BroadcastReceiver` itself (e.g., `abortBroadcast()`) but want to do on a background thread. You need your background thread to have access to the `PendingResult` — in this case, via a `final` local variable. When you are done with your work, call `finish()` on the `PendingResult`.

If you click the "Send Async Broadcast" button, even though we are still sleeping for 7 seconds, we are doing so on a background thread, and so our user interface is still responsive.

Saving SharedPreferences

The classic way to save `SharedPreferences.Editor` changes was via a call to `commit()`. This writes the preference information to an XML file on whatever thread you are on — another hidden source of disk I/O you might be doing on the main application thread.

If you are on API Level 9, and you are willing to blindly try saving the changes, use the new `apply()` method on `SharedPreferences.Editor`, which works asynchronously.

If you need to support older versions of Android, or you really want the boolean return value from `commit()`, consider doing the `commit()` call in an `AsyncTask` or background thread.

And, of course, to support both of these, you will need to employ tricks like conditional class loading. You can see that used for saving `SharedPreferences` in the CPU-Java/PrefsPersist sample project. The activity reads in a preference, puts the current value on the screen, then updates the preference with the help of an `AbstractPrefsPersistStrategy` class and its `persist()` method:

```
package com.commonsware.android.tuning.prefs;

import android.app.Activity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.widget.TextView;

public class PrefsPersistActivity extends Activity {
```

```
  private static final String KEY="counter";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(this);
    int counter=prefs.getInt(KEY, 0);

    ((TextView)findViewById(R.id.value)).setText(String.valueOf(counter));

    AbstractPrefsPersistStrategy.persist(prefs.edit().putInt(KEY, counter+1));
  }
}
```

(from CPU-Java/PrefsPersist/app/src/main/java/com/commonsware/android/tuning/prefs/PrefsPersistActivity.java)

`AbstractPrefsPersistStrategy` is an abstract base class that will hold a strategy implementation, depending on Android version. On pre-Honeycomb builds, it uses an implementation that forks a background thread to perform the `commit()`:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences;
import android.os.Build;

abstract public class AbstractPrefsPersistStrategy {
  abstract void persistAsync(SharedPreferences.Editor editor);

  private static final AbstractPrefsPersistStrategy INSTANCE=initImpl();

  public static void persist(SharedPreferences.Editor editor) {
    INSTANCE.persistAsync(editor);
  }

  private static AbstractPrefsPersistStrategy initImpl() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<Build.VERSION_CODES.HONEYCOMB) {
      return(new CommitAsyncStrategy());
    }

    return(new ApplyStrategy());
  }
```

```
  static class CommitAsyncStrategy extends AbstractPrefsPersistStrategy {
    @Override
    void persistAsync(final SharedPreferences.Editor editor) {
      (new Thread() {
        @Override
        public void run() {
          editor.commit();
        }
      }).start();
    }
  }
}
```

(from CPU-Java/PrefsPersist/app/src/main/java/com/commonsware/android/tuning/prefs/AbstractPrefsPersistStrategy.java)

On Honeycomb and higher, it uses a separate strategy class that uses the new `apply()` method:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences.Editor;

public class ApplyStrategy extends AbstractPrefsPersistStrategy {

  @Override
  void persistAsync(Editor editor) {
    editor.apply();
  }
}
```

(from CPU-Java/PrefsPersist/app/src/main/java/com/commonsware/android/tuning/prefs/ApplyStrategy.java)

By separating the Honeycomb-specific code out into a separate class, we can avoid loading it on older devices and encountering the dreaded `VerifyError`.

Whether using the built-in `apply()` method is worth dealing with multiple strategies, versus simply calling `commit()` on a background thread, is up to you.

Improve Throughput and Responsiveness

Being efficient and doing work on the proper thread may still not be enough. It could be that your work is not consuming excessive CPU time, but is taking too long in "wall clock time" (e.g., the user sits waiting too long at a `ProgressDialog`). Or, it could be that your work, while efficient and in the background, is causing difficulty for foreground operations.

The following sections outline some common problems and solutions in this area.

Minimize Disk Writes

Earlier in this book, we emphasized moving disk writes off to background threads.

Even better is to get rid of some of the disk writes entirely.

A big culprit here comes in the form of database operations. By default, each `insert()`, `update()`, or `delete()`, or any `execSQL()` invocation that modifies data, will occur in its own transaction. Each transaction involves a set of disk writes. Many times, this is not a problem. But, if you are doing a lot of these – such as importing records from a CSV file — hundreds or thousands of transactions will mean thousands of individual disk writes, and that can take some time. You may wish to wrap those operations in your own transaction, using methods like `beginTransaction()`, simply to reduce the number of transactions and, therefore, disk writes.

If you are doing your own disk I/O beyond databases, you may encounter similar sorts of issues. Overall, it is better to do a few larger writes than lots of little ones.

Set Thread Priority

Threads you fork, by default, run at a default priority: `THREAD_PRIORITY_DEFAULT` as defined on the `Process` class. This is a lower priority than the main application thread (`THREAD_PRIORITY_DISPLAY`).

Threads you use via `AsyncTask` run at a lower priority (`THREAD_PRIORITY_BACKGROUND`). If you fork your own threads, then, you might wish to consider moving them to a lower priority as well, to affect how much time they get compared to the main application thread. You can do this via `setThreadPriority()` on the `Process` class.

The lowest possible priority, `THREAD_PRIORITY_LOWEST`, is described as “only for those who really, really don’t want to run if anything else is happening”. You might use this for “idle-time processing”, but bear in mind that the thread will be paused a lot to allow other threads to run.

Lower-priority threads will help ensure that your background work does not affect your foreground UI. Processes themselves are put in a lower-priority class as they

move to the background (e.g., you have no activities visible), which further reduces the amount of CPU time you will be using at any given moment.

Also, note that `IntentService` uses a thread at default (*not* background) priority — you may wish to drop the priority of this thread to something that will be lower than your main application thread, to minimize how much CPU time the `IntentService` steals from your UI.

Do the Work Some Other Time

Just because you could do the work now does not mean you should do the work now. Perhaps a better answer is to do the work later, or do part of the work now and part of the work later.

For example, suppose that you have your own database of points of interest for your custom map application. Periodically, you publish a new database on your Web site, which your Android app should download. Odds are decent that the user is not in desperate need for this new database right away. In fact, the CPU time and disk I/O time to download and save the database might incrementally interfere with the foreground application, despite your best efforts.

In this case, not only should you check for and download the database when the user is unlikely to be using the device (e.g., before dawn), but you should check whether the screen is on via `isScreenOn()` on `PowerManager`, and delay the work to sometime when the screen is off. For example, you could have `AlarmManager` set up to have your code check for updates every 24 hours at 4am. If, at 4am, the screen is on, your code could skip the download and wait until tomorrow, or skip the download and add a one-shot alarm to wake you up in 30 minutes, in hopes that the user will no longer be using the device.

At the same time, you may wish to consider having a “refresh” menu choice somewhere, for when the user specifically wants you to go get the update (if available) *now*, for whatever reason.

Finding and Eliminating Jank

A user interface is considered "janky" if it stutters or otherwise fails to operate smoothly, particularly during animated effects like scrolling. Finding and eliminating the causes of janky behavior ("jank") is part science, part art, and part throwing darts at a dartboard.

This chapter will outline some techniques for identifying and removing jank from a user interface. The steps shown here originated in a blog post by Google's Romain Guy, with a few additional twists and turns due to the different nature of the particular case being studied. Mr. Guy's blog post is essential reading for all advanced Android developers, and the author is deeply indebted to Mr. Guy for his work in this area.

Prerequisites

The only hard prerequisite for this chapter is having read the core chapters and the chapter on finding CPU bottlenecks.

That being said, having read the chapter on animators would help understand portions of this chapter a bit better.

The Case: ThreePaneDemoBC

In the chapter on animators, we examined an implementation of the Gmail-style three-pane layout with animated transitions (a.k.a., "The Three-Fragment Problem"). The implementation shown there originated with a Stack Overflow question with the solution presented in this book offered as an answer.

A commenter on that answer pointed out that he detected some stutter, even on decent hardware.

This chapter reviews the steps that were taken to determine if we really are doing things incorrectly, what specifically we are doing wrong, and what can be done to fix it.

Are We Janky?

In the eyes of this book's author, the three-pane implementation presented in the chapter on animators was perfectly reasonable on good hardware.

There are two lessons to take from this:

1. It is better to come up with an objective definition for "jank" and test to see if your code meets that definition at various points
2. The author of this book is very tolerant of janky user interfaces

The results shown in the chapter on CPU measurement for the `gfxinfo` and `systrace` tools comes from the three-pane demo code. The `gfxinfo` and the `systrace` results both point to the three-pane demo spending too much time doing work and therefore dropping some number of frames. This lines up with the visual report, and indicates that we have some work to do to try to improve matters.

Finding the Source of the Jank

Just because we know that we are janky does not mean that we have any idea what to do about it. We need to conduct some further analysis to determine where, exactly, our jank is coming from.

Traceview

One thing that we can do to help further refine the source of our trouble is to use Traceview. As outlined in the section on Traceview, Traceview reports how many calls were made of various methods in our code (and in the framework code) and how much time was spent there.

Here are some of the results from a Traceview run on the three-pane demo on a Nexus 7:

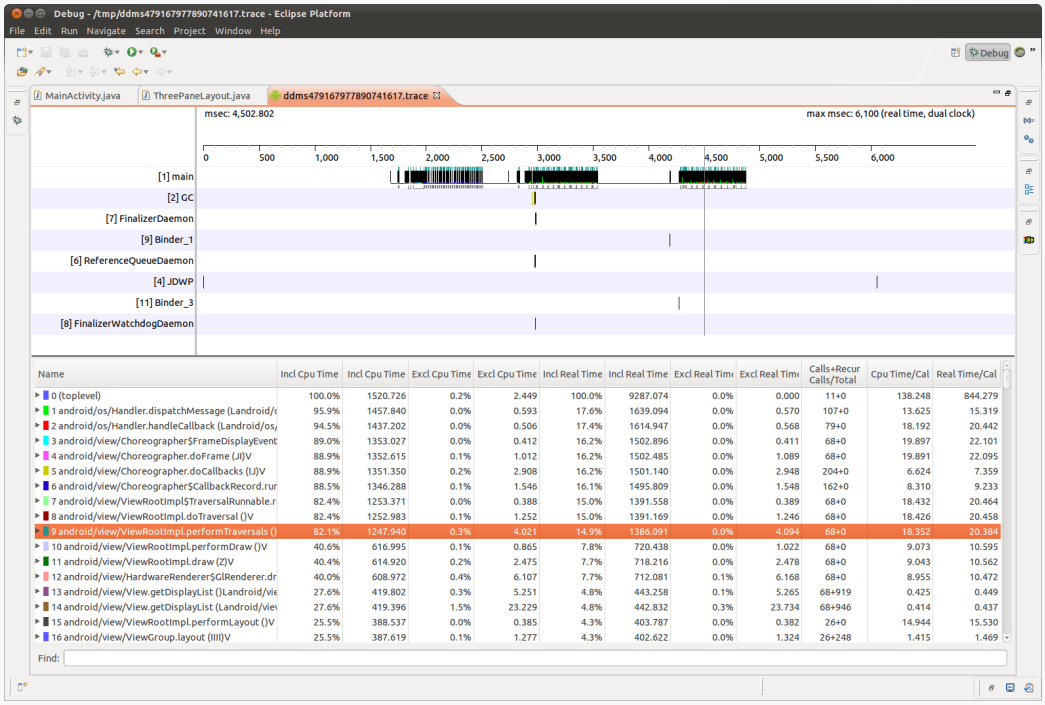


Figure 1002: Traceview of Three-Pane Demo

We see that 88.9% of our CPU time is spent in `doFrame()` on `Choreographer` and the calls triggered from it. `doFrame()` is a private method which, as the name suggests, performs the drawing, processing, and executing of a single frame's worth of rendering. More importantly, we see that `doFrame()` was called 68 times during our test run, meaning that our UI changed 68 times during the ~3 seconds of activity during our trace.

Further down the table, we see that `layout()` on `ViewGroup` was called 26 times directly (and 248 more times via recursion), contributing about 25% of the time consumed by `doFrame()`. Since `layout()` is called on less than half of the `doFrame()` calls, the time consumed by `layout()` makes up a fairly significant portion of the `doFrame()` time during those 26 frames.

More importantly, `layout()` is something that *we* trigger. It implies that we have made some change to our UI content that requires a layout pass of some `ViewGroup`.

Having a layout pass on occasion is perfectly normal, particularly in response to user input. A `layout()` might be triggered by the user tapping on a row in one of our `ListViews`, for example. But we are not doing 26 user input events in our test — all we are doing is tapping one time each on a pair of `ListView` rows, then pressing the

BACK button. This implies that something else in our code is causing `layout()` to be needed.

Unfortunately, at this point, Traceview does not help much, because the calls to `layout()` are asynchronous with respect to our own code, so it will not be all that obvious where the extra calls are coming from. This is where we need some expert help, as we will see later in this chapter.

Overdraw

Another common source of jank is overdraw. Overdraw refers to the act of painting the same pixel several times, due to overlapping components. For example:

- The activity window itself has a background
- You have a container that fills the activity's content, such as a `ListView`, which has a background
- You have children in that container with backgrounds (row), who have their own children with backgrounds and, eventually, content (widgets like `ImageView` and `TextView`)

Places where there is overlap, the OS might set the color of a pixel several times per frame, wasting time.

The easiest way to track down overdraw is to use the "Show GPU overdraw" option in the Developer Options portion of the Settings app:

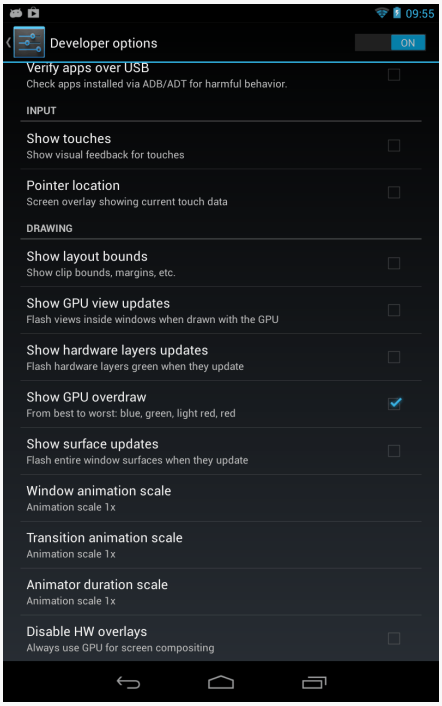


Figure 1003: Nexus 7 Developer Options, with "Show GPU overdraw"

This option is only available on Android 4.2 and higher.

When you enable this option, then restart your app's process (if it was already running), Android will shade pixels that are overdrawn:

- Blue for pixels that are drawn twice
- Green for pixels that are drawn three times
- Pink for pixels that are drawn four times
- Red for pixels that are drawn five or more times

In short: pink and red are bad. Green and blue are OK, though if you have large patches of either shade, you might consider trying to see if there's a way to get rid of the overdraw.

Of course, the fact that these are shades applied to existing pixel colors may make it a bit difficult to tell exactly where the overdraw is occurring. For example, a red portion of your UI might be red from overdraw... or it might be red because you made it red. Temporarily changing your color scheme to something else (e.g., yellow) will help distinguish what is overdraw and what is just the natural UI coloration.

If you enable this option on a Nexus 7 and run the three-pane demo, you will see very little blue or green (beyond the normal blue of the activated state of our `ListView` rows), and virtually no red:

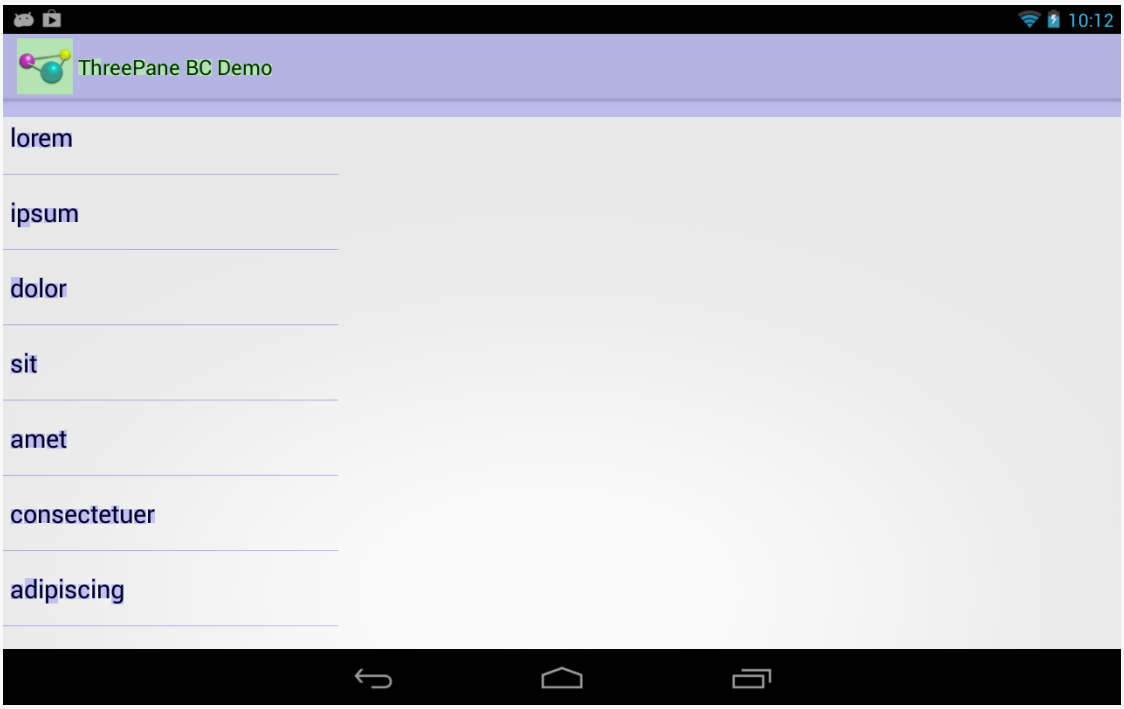


Figure 1004: Three-Pane Demo, As Initially Launched, Showing Overdraw

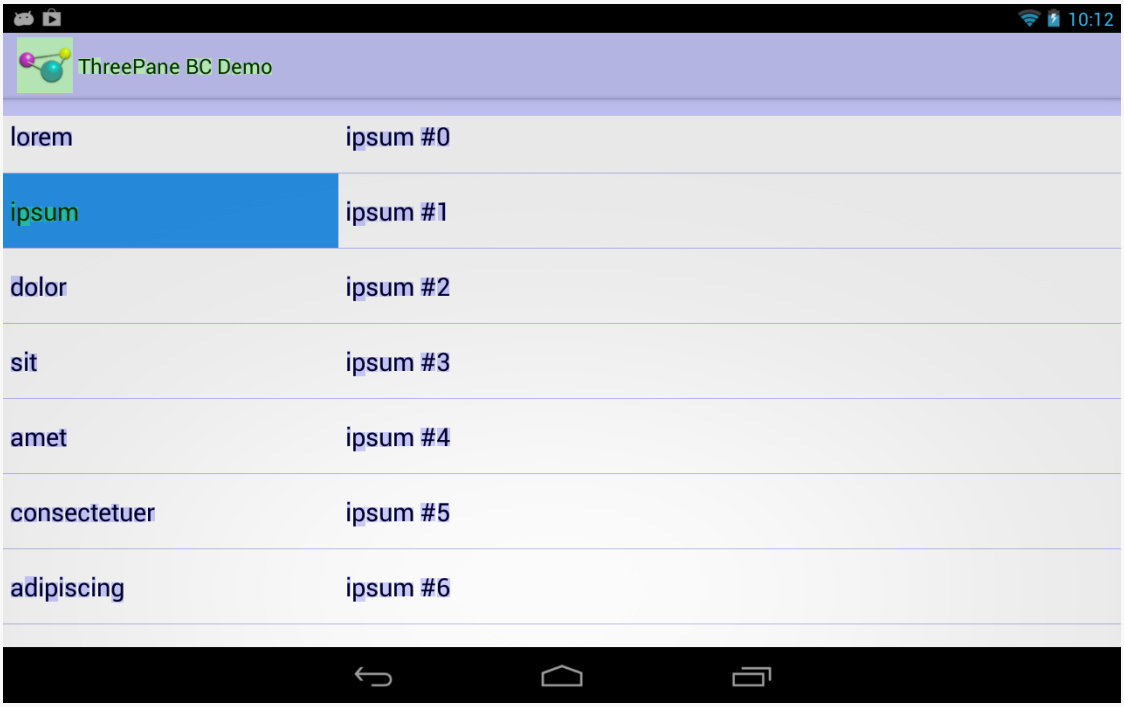


Figure 1005: Three-Pane Demo, Left and Middle Panes, Showing Overdraw

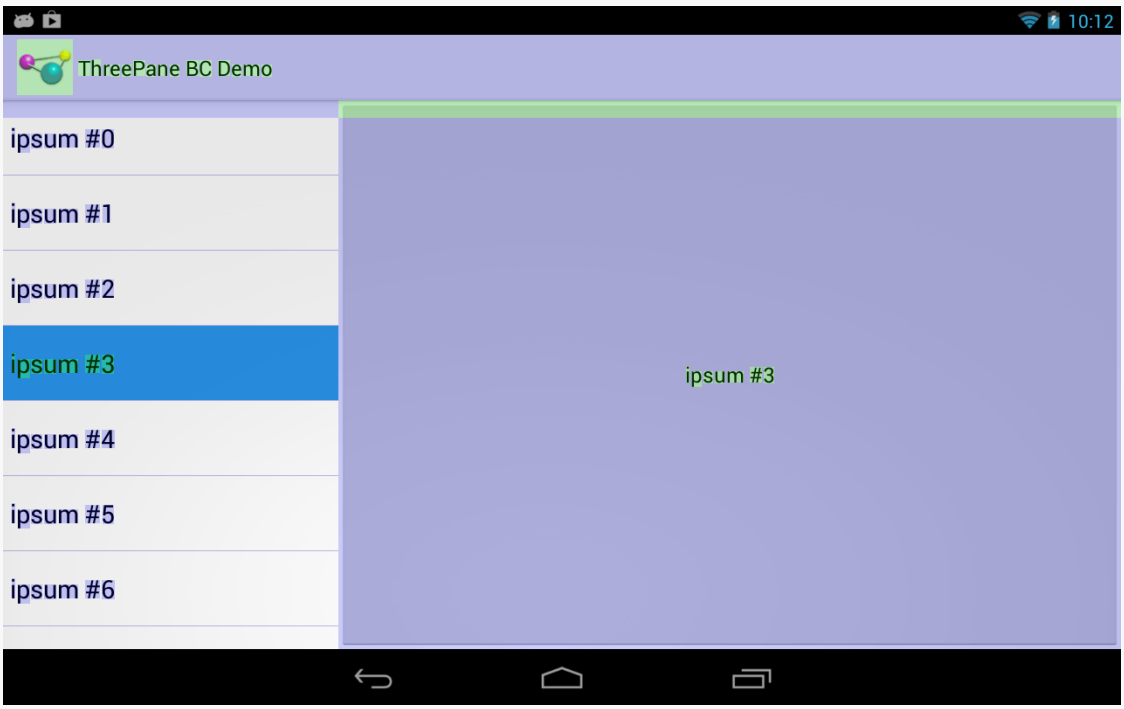


Figure 1006: Three-Pane Demo, Middle and Right Panes, Showing Overdraw

Figure 1007: Three-Pane Demo, Left and Middle Panes Via BACK, Showing Overdraw

On the other hand, bringing up the Contacts app on the same Nexus 7 shows significantly more overdraw:

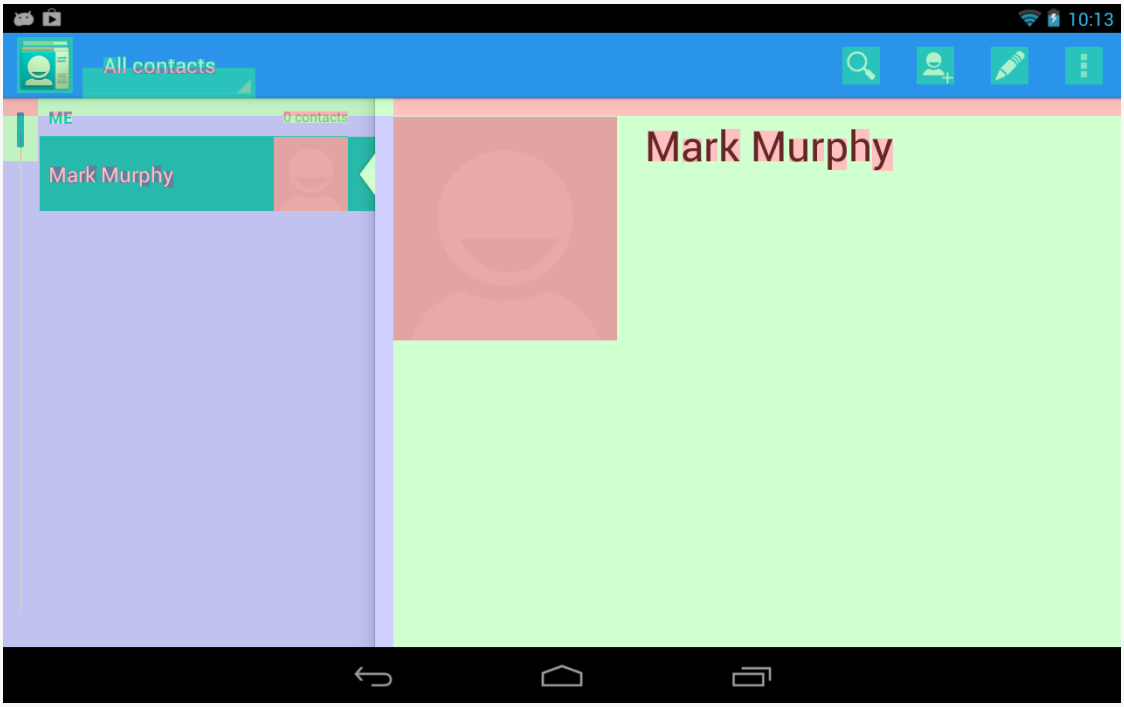


Figure 1008: Contacts App, Showing Overdraw

The good news is that our app is not suffering performance problems due to overdraw.

The bad news is that the Contacts app is.

The good news is that if you are reading this, you are probably not responsible for maintaining the Contacts app.

The Contacts app's major problems come from the contact photos, or placeholders as seen here. Either the `ImageView` has a background, or the `ImageView` fills some container with a background. For example, the `ImageView` might be in some container with a background to provide a bevel effect around the image. Making the portion of the background that is behind the `ImageView` be transparent will eliminate the overdraw.

Note: some GPU architectures can automatically fix overdraw in select places, while others cannot. Notably, the Tegra 3 cannot. Hence, the Tegra 3 is a good test platform for using this overdraw-detection feature of Android.

Extraneous Views

Another related source of jank is having too many extraneous views. Each widget and container contributes to the cost of drawing the overall UI, so having extraneous views adds overhead.

Perhaps the most common scenario for extraneous views is the single-child container. If a container will only ever hold one child, perhaps you can get rid of that container. Not only will this speed up execution at runtime, but it can help avoid running out of stack space.

One way to find these extraneous views is to bring up your user interface in the Android Studio Layout Inspector, or in the older Hierarchy View tool.

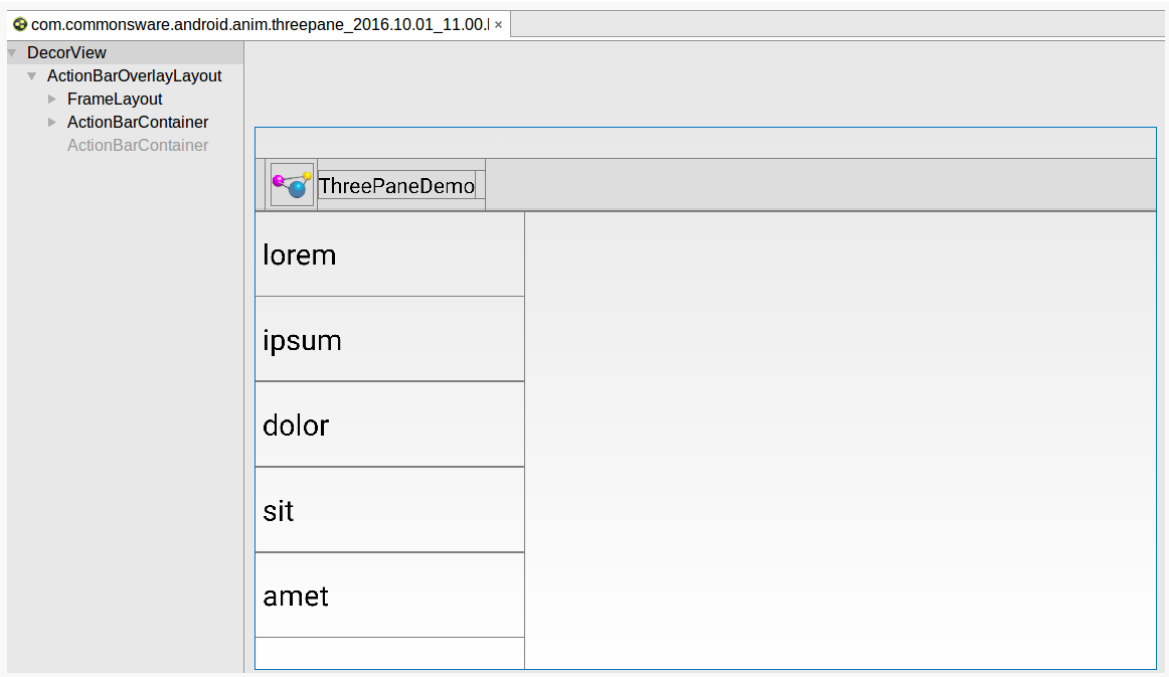


Figure 1009: Three-Pane Demo, As Initially Launched, in Layout Inspector

The root `DecorView` has only one child. This is a framework-supplied container, and so we have no control over it.

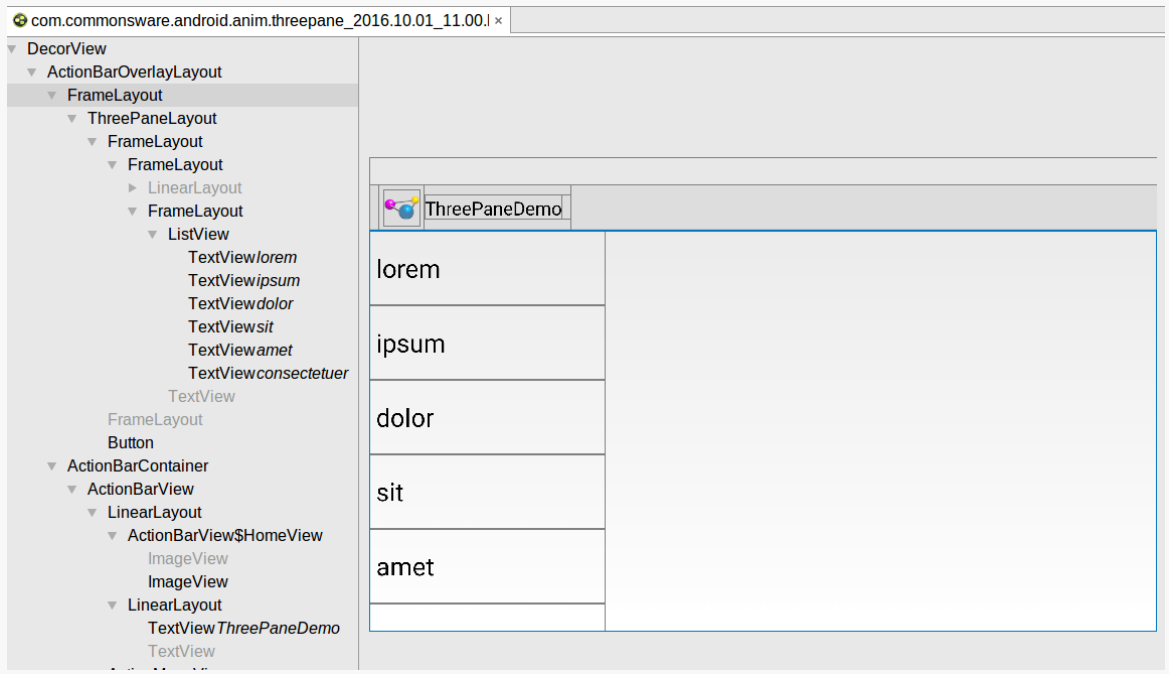


Figure 1010: Three-Pane Demo in Layout Inspector, Showing More Hierarchy

Here, we have a `FrameLayout` holding onto just one child, our `ThreePaneLayout` custom view. We set up `ThreePaneLayout` as being our activity's content view. The "content view" of an activity is poured into a `FrameLayout`, supplied by the Android framework — that is the `FrameLayout` seen in Hierarchy View. We have no good way to get rid of this `FrameLayout`. Fortunately, `FrameLayout` is a very cheap container, in terms of runtime execution speed.

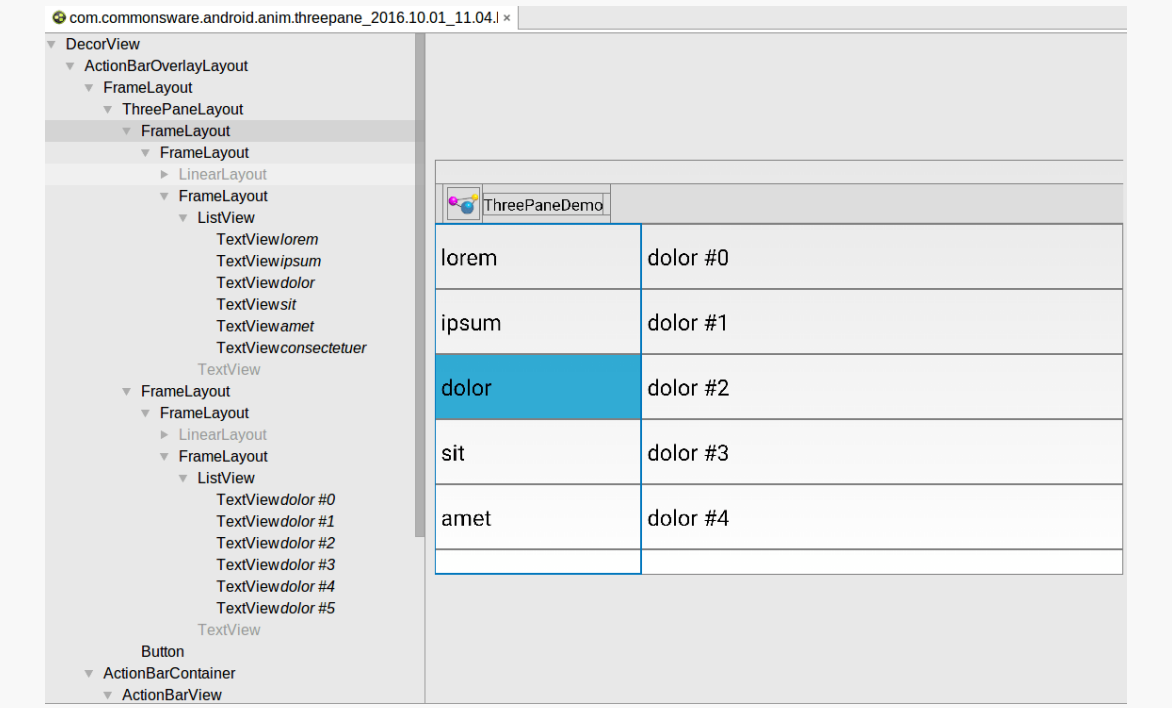


Figure 1011: More Single-Child Containers in Three-Pane Demo, from Layout Inspector

Here, we see that our left and middle `FrameLayout` containers, for our left and middle panes, each contain one child, a `FrameLayout`. These come from using `ListFragment`, as does the `LinearLayout` (having visibility `GONE`) and another `FrameLayout`, before getting to the `ListView`. Short of writing our own fragment for holding a `ListView`, there is nothing we can do about these extraneous views.

Conclusion: Too Many layout() Calls?

Given that overdraw does not seem to be a problem and that we have few extraneous views under our control, it would seem that perhaps we should return our attention to the extra `layout()` calls. While trying to get rid of the `ListFragment` extraneous views would make those `layout()` calls incrementally cheaper, we will get more value by getting rid of the unnecessary calls in the first place, if indeed they are unnecessary.

Where Things Went Wrong

Of course, it doesn't hurt to call in an expert, to try to confirm exactly what is going on.

Chet Haase — Google engineer on Android, celebrated book author, and part-time comedian – chimed in with an answer to a Stack Overflow question about this three-pane animation, asked by the person who commented about the dropped frames on the original Stack Overflow question.

The key statement from his answer was:

> Sliding things around is fine (translationX/Y), fading things in/out is good (alpha), but actually laying things out on every frame? Just say no.

Specifically, he is referring to our use of `ObjectAnimator` to change the width of the middle pane as we show and hide the right pane. Each time we change the width of the middle pane, we trigger a `layout()` call, to reposition the child widgets within that pane as needed. Our animations are adding ~20 `layout()` calls, introducing overhead that is pushing us over the per-frame limit on the Nexus 7.

Removing the Jank

To remove the jank, we need to remove the `ObjectAnimator` changing the width of the middle pane on the fly. You can see the results of this in the `Jank/ThreePaneBC` sample app.

Now, our `showLeft()` and `hideLeft()` methods immediately change the width of the middle pane, rather than arranging its animation:

```
public void hideLeft() {
  if (leftWidth == -1) {
    leftWidth=left.getWidth();
    middleWidthNormal=middle.getWidth();
    resetWidget(left, leftWidth);
    resetWidget(middle, middleWidthNormal);
    resetWidget(right, middleWidthNormal);
    requestLayout();
  }

  translateWidgets(-1 * leftWidth, left, middle, right);
  setMiddleWidth(leftWidth);
```

```
}

public void showLeft() {
  translateWidgets(leftWidth, left, middle, right);
  setMiddleWidth(middleWidthNormal);
}

private void setMiddleWidth(int value) {
  middle.getLayoutParams().width=value;
  requestLayout();
}
```

(from Jank/ThreePaneBC/app/src/main/java/com/commonsware/android/anim/threepane/ThreePaneLayout.java)

This does not provide nearly as good of a UI as the original. However, the revised solution does reduce the jank, as seen in this `gfxinfo` output:

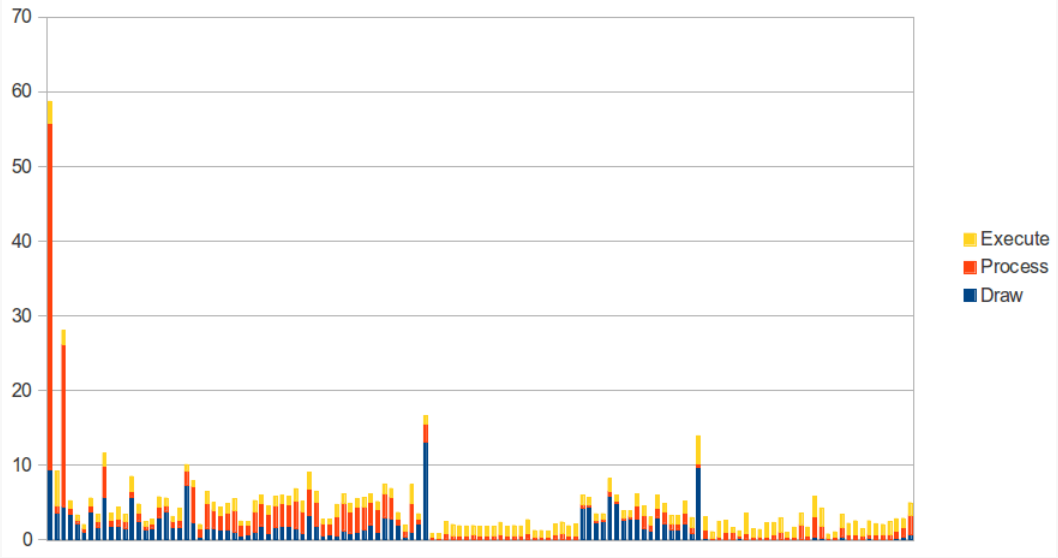


Figure 1012: gfxinfo Output of Revised ThreePaneDemoBC

There are probably ways to improve upon the revised jank-free implementation. Lacking that, it is up to you to decide if the amount of jank found in the original implementation is worth the improved animation or not.

Frame Metrics API

Above, we looked at `gfxtrace`, which gives you a dump of information related to the rendering of your UI, including how much time is spent in different aspects of that

work. However, `gfxtrace` is a device-wide data collection, and so there may be a bit of noise in the results due to other things being updated (`Notification` icons, other status bar icons, status bar clock, third-party app overlay views, etc.). Plus, `gfxtrace` itself represents a chunk of overhead, slowing down the device.

Android 7.0+ gives us a frame metrics API. We can register a listener to find out the same sorts of information that `gfxtrace` reports, but only to our app, and in Java code. We can log that to Logcat, or do our own diagnostic overlay, or cache it and later send it to a server for analysis. We can also elect to collect this data in our instrumentation tests, to help detect a significant shift in our UI rendering performance.

What Data You Get

Unfortunately, there is virtually no documentation on the data that we get.

We will receive `FrameMetrics` objects, on which we can call `getMetric()` and provide the identifier of one of the specific metrics that we can retrieve. Note that `FrameMetrics` objects get recycled, so when your listener is handed one, you should copy the data you want out of it and save it somewhere else, rather than attempt to hold onto the `FrameMetrics` object itself. This pattern is used elsewhere in Android, such as with sensors.

Most of the metrics represent amounts of time, measured in nanoseconds, returned as `long` values:

Metric ID	Role
`ANIMATION_DURATION`	Time spent in animation callbacks
`COMMAND_ISSUE_DURATION`	Time spent issuing draw commands to the GPU
`DRAW_DURATION`	Time spent building display lists of commands
`INPUT_HANDLING_DURATION`	Time spent in input-handling callbacks
`LAYOUT_MEASURE_DURATION`	Time spent in `measure()` and `layout()` work for your view hierarchy
`SWAP_BUFFERS_DURATION`	Time spent sending the frame buffer for this frame to the display
`SYNC_DURATION`	Time spent synchronizing the display lists with the render thread
`UNKNOWN_DELAY_DURATION`	Time spent waiting for the UI thread to process the frame

There is also `TOTAL_DURATION`, which should match the sum of the above values. And, there is `FIRST_DRAW_FRAME`, which returns a boolean value encoded as a `long` (`0` for false, `1` for true), indicating whether this frame was the first to draw in a new `Window` layout.

Developers with deep understanding of the graphics rendering pipeline in Android may recognize these values. For everybody else, the objective is for them to be as low as possible, and for the sum to be well under the 16ms (16,000,000ns) we have for a frame.

How You Get That Data

The `Jank/FrameMetrics` sample project is another variation on the "show a list of videos in a `RecyclerView`" sample that was profiled in the chapter on `RecyclerView` and elsewhere.

This time, though, we have a menu resource:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/record"
    android:icon="@drawable/ic_record_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_record" />
  <item
    android:id="@+id/stop"
    android:icon="@drawable/ic_stop_white_24dp"
    android:showAsAction="always"
    android:title="@string/menu_stop"
    android:visible="false" />
</menu>
```

(from Jank/FrameMetrics/app/src/main/res/menu/actions.xml)

It defines two action bar items: one to record frame metrics, and one to stop the recording, whether the latter is initially invisible.

In `MainActivity`, we inflate that menu resource, find those two action items, and hold onto them in fields, via `onCreateOptionsMenu()`:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);
```

```
    record=menu.findItem(R.id.record);
    stop=menu.findItem(R.id.stop);

    return(super.onCreateOptionsMenu(menu));
```

(from Jank/FrameMetrics/app/src/main/java/com/commonsware/android/jank/framemetrics/MainActivity.java)

In `onOptionsItemSelected()`, when one of those two is tapped, we invert their visibilities, so when the user taps `record`, the `stop` item appears, and vice versa:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.record) {
    record.setVisible(false);
    stop.setVisible(true);
    afm=new AggregateFrameMetrics();
    getWindow()
      .addOnFrameMetricsAvailableListener(this,
        new Handler(handlerThread.getLooper()));

    return(true);
  }
  else if (item.getItemId()==R.id.stop) {
    record.setVisible(true);
    stop.setVisible(false);
    getWindow().removeOnFrameMetricsAvailableListener(this);
    afm.log(getClass().getSimpleName());
    afm=null;

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from Jank/FrameMetrics/app/src/main/java/com/commonsware/android/jank/framemetrics/MainActivity.java)

However, in addition, when the user taps `record`, we:

- Create an instance of an `AggregateFrameMetrics` class and hold onto it in a field (`afm`)
- Call `addOnFrameMetricsAvailableListener()` on the activity's `Window`, passing in a `Window.OnFrameMetricsAvailableListener` implementation (here, our `MainActivity` itself) and a `Handler` on which we want to have that listener's callback method be invoked (here, a `Handler` tied to a `HandlerThread` created when the activity is created)

The result is that when the user taps `record`, our activity's `onFrameMetricsAvailable()` method will be called on that `HandlerThread`, providing us with metrics on the just-completed frame:

```
  @Override
  public void onFrameMetricsAvailable(Window window,
                                      FrameMetrics frameMetrics,
                                      int droppedEvents) {
    afm.add(frameMetrics, droppedEvents);
  }
```

(from Jank/FrameMetrics/app/src/main/java/com/commonsware/android/jank/framemetrics/MainActivity.java)

`onFrameMetricsAvailable()` is passed three parameters:

- the `Window` which we registered for metrics on
- the `FrameMetrics` for that frame
- an `int` indicating how many `onFrameMetricsAvailable()` calls were missed because we took too long in the last `onFrameMetricsAvailable()` call processing the previous results (ideally, this is always 0)

Here, we pass the metrics and dropped-events values to the `AggregateFrameMetrics` object.

Back up in `onOptionsItemSelected()`, if the user taps on `stop`, we remove our activity as a listener via `removeOnFrameMetricsAvailableListener()` and call `log()` on the `AggregateFrameMetrics` object.

`AggregateFrameMetrics`, in turn, is responsible for maintaining a running total of the various individual metrics and dumping them to Logcat when `log()` is called:

```
package com.commonsware.android.jank.framemetrics;

import android.util.Log;
import android.view.FrameMetrics;

public class AggregateFrameMetrics {
  int droppedReports;
  long animationDuration;
  long commandIssueDuration;
  long drawDuration;
  long inputHandlingDuration;
  long layoutMeasureDuration;
  long swapBuffersDuration;
  long syncDuration;
```

```
  long unknownDelayDuration;
  long totalDuration;

  void add(FrameMetrics metrics, int droppedReports) {
    this.droppedReports+=droppedReports;

    animationDuration+=
      metrics.getMetric(FrameMetrics.ANIMATION_DURATION);
    commandIssueDuration+=
      metrics.getMetric(FrameMetrics.COMMAND_ISSUE_DURATION);
    drawDuration+=metrics.getMetric(FrameMetrics.DRAW_DURATION);
    inputHandlingDuration+=
      metrics.getMetric(FrameMetrics.INPUT_HANDLING_DURATION);
    layoutMeasureDuration+=
      metrics.getMetric(FrameMetrics.LAYOUT_MEASURE_DURATION);
    swapBuffersDuration+=
      metrics.getMetric(FrameMetrics.SWAP_BUFFERS_DURATION);
    syncDuration+=metrics.getMetric(FrameMetrics.SYNC_DURATION);
    unknownDelayDuration+=
      metrics.getMetric(FrameMetrics.UNKNOWN_DELAY_DURATION);
    totalDuration+=metrics.getMetric(FrameMetrics.TOTAL_DURATION);
  }

  void log(String tag) {
    Log.d(tag, String.format("animation: %dns", animationDuration));
    Log.d(tag, String.format("command issue: %dns", commandIssueDuration));
    Log.d(tag, String.format("draw: %dns", drawDuration));
    Log.d(tag, String.format("input handling: %dns", inputHandlingDuration));
    Log.d(tag, String.format("layout measure: %dns", layoutMeasureDuration));
    Log.d(tag, String.format("swap buffers: %dns", swapBuffersDuration));
    Log.d(tag, String.format("sync: %dns", syncDuration));
    Log.d(tag, String.format("unknown: %dns", unknownDelayDuration));
    Log.d(tag, String.format("total: %dns", totalDuration));
    Log.d(tag, String.format("%d dropped reports", droppedReports));
  }
}
```

(from Jank/FrameMetrics/app/src/main/java/com/commonsware/android/jank/framemetrics/AggregateFrameMetrics.java)

The net effect is that between taps on `record` and `stop`, we collect information about the UI rendering performance. If you try this on a device that has a bunch of videos on it, scroll through the list while the collection is ongoing. Once you click `stop` to end the collection, the aggregated results are printed to Logcat.

Issues with Bandwidth

As anyone who owned an Apple Newton or Palm V PDA back in the 1990's knows, handheld devices have been around for quite some time. For a very long time, they were a niche product, associated with geeks, nerds, and the occasional business executive.

Internet access changed all of that.

Blackberry for enterprise messaging — an outgrowth of its original two-way paging approach — blazed part of the trail, but the concept "crossed the chasm" to ordinary people with the advent of the iPhone, Android devices, and similar equipment.

Therefore, it is not terribly surprising when Android developers want to add Internet capabilities to their apps. To the contrary, it is almost unusual when you encounter an app that does *not* want to use the Internet for something or another.

However, mobile Internet access inherits all of the classic problems of Internet access (e.g., "server not found") and adds new and exciting challenges, all of which can leave a developer with an app that has performance issues.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

You're Using Too Much of the Slow Stuff

To paraphrase America's Founding Fathers, "all Internet connections are not created equal".

One form of inequality is speed. Different classes of connection have different theoretical upper bounds. WiMAX and other 4G connections are theoretically faster than 3G connections, which are theoretically faster than 2G or EDGE connections. WiFi is theoretically ridiculously fast though it is typically limited by the ISP connection, and ISP connections can run the gamut from really fast to merely good.

However, "theoretical" bounds tend to run afoul of reality. There are plenty of places where high-speed mobile data connections are non-existent, despite what the carriers' coverage maps claim. 2G mobile data works, but is not especially speedy. This layers on top of the typical Internet congestion issues, along with typically transitory problems (e.g., trying to get connectivity while attending a technology conference keynote presentation).

Beyond that, there are financial issues. While WiFi is usually unmetered (no incremental cost per MB/GB), many mobile data connections are metered. Those mobile data connections that are not metered in theory – advertised as "unlimited" — have usage caps that, once exceeded, impose costs or impose speed limits.

Hence, what runs quickly in the lab may run much more slowly in users' hands.

If you followed the instructions in previous chapters on CPU bottlenecks, the limited bandwidth will not cause your UI to become "janky", in that it will be responsive to touches and taps. However, poor connectivity will mean that you are simply slow to respond to user requests. For example, clicking the "check for new email" menu button has no immediate effect. If you feel that you need a splash screen or progress indicator to tell the user that "we are really checking for new email, honest", then you know that your Internet access is slower than is ideal.

Obviously, some of this is unavoidable. However, the objective of the chapters in this part of the book is to give you an idea of ways to reduce your bandwidth consumption, making those delays be that much less annoying for your users.

You're Using Too Much of the Expensive Stuff

Mobile data tends to come with more strings attached than does WiFi.

In the US, it used to be that mobile data connections included unlimited usage. Now, at best, a mobile data plan has "unlimited" usage for a curious definition of the term "unlimited". More and more carriers are moving towards a hard cap — go above the cap, and you either cannot use more bandwidth, have your speeds curtailed, or pay significantly for additional bandwidth.

Outside of the US, the "pay significantly for bandwidth" approach is fairly typical. So-called "metered" data plans simply charge you such-and-so per MB or GB of bandwidth.

And, to top it off, roaming almost always is a metered plan. So, a US resident traveling overseas, even with a SIM and phone that supports international usage, would pay a ridiculous sum for bandwidth. Stories of phone bills in the tens of thousands of dollars abound, where people simply used their phone as they normally would when they were outside of their home network.

Hence, if you use a fair bit of bandwidth, it would be really nice if you offered users means to consume less of it when they are on mobile data compared to WiFi (which is typically unmetered). You could elect to poll your server less frequently, for example, giving the users the ability to specify separate polling periods depending on which type of connection they have.

And, of course, there are other "costs" for using bandwidth besides direct monetary costs. For example, downloading data over a slower mobile data connection may consume more power than downloading the same data over WiFi — while the WiFi radio might consume additional power, the time difference might account for more power consumption, if the CPU could be powered down for the rest of that time.

These chapters will show you how you can react to changes in connectivity and approaches for how to use that information to reduce costs for the user.

You're Using Too Much of Somebody Else's Stuff

It is easy for developers to think that they alone are using a user's device. Alas, this is infrequently the case, particularly when it comes to background Internet access.

While your application is busily downloading stuff, some other application might be busily downloading stuff. In principle, this should not be an issue, as multiple applications can access the Internet simultaneously. However, bandwidth can become an issue. If you are in the background, and the other application is in the

foreground, the user might notice that bandwidth is an issue. For example, users might be unhappy if your downloads are impeding their ability to watch streaming video, or play their favorite Android-based MMORPG, or whatever.

A polite Android application will test to see whether the foreground application is heavily using the Internet and will curtail its own Internet use while that is going on. This chapter will help you learn how to make that determination and how to respond.

You're Using Too Much… And There Is None

Not only might location dictate how much bandwidth you have, but whether you have any bandwidth at all.

While some people think that the entire planet has connectivity, reality once again dictates otherwise. Major metropolitan areas have connectivity. Outlying areas are much more hit-or-miss. Voice is sometimes a challenge, let alone data. And it only *seems* as though there is a Starbucks every 100 meters in the US, which might actually provide blanket WiFi coverage.

Then, of course, there are planes (many still do not offer in-flight WiFi at this time), international travel without an international-capable phone plan, and so on.

Some Android applications have the potential to still offer near-complete functionality despite this, with a bit of user assistance. For example, Google Maps for Android has an offline caching feature, which will download data for a 10-mile radius from a given point, for use while the device is otherwise offline.

Here, the issue becomes less one of bandwidth (other than detecting that you have no connection) and more one of caching and storage. The space-related issues that these techniques can raise will be covered elsewhere in this book.

Focus On: TrafficStats

To be able to have more intelligent code — code that can adapt to Internet activity on the device — Android offers the `TrafficStats` class. This class really is a gateway to a block of native code that reports on traffic usage for the entire device and per-application, for both received and transmitted data. This chapter will examine how you can access `TrafficStats` and interpret its data.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

TrafficStats Basics

The `TrafficStats` class is not designed to be instantiated — you will not be invoking a constructor by calling `new TrafficStats()` or something like that. Rather, `TrafficStats` is merely a collection of static methods, mapped to native code, that provide access to point-in-time traffic values. No special permissions are needed to use any of these methods. Most of the methods were added in API Level 8 and therefore should be callable on most Android devices in use today.

Device Statistics

If you are interested in overall traffic, you will probably care most about the `getTotalRxBytes()` and `getTotalTxBytes()` on `TrafficStats`. These methods return received and transmitted traffic, respectively, measured in bytes.

You also have:

1. `getTotalRxPackets()` and `getTotalTxPackets()`, if for your case measuring IP packets is a better measure than bytes
2. `getMobileRxBytes()` and `getMobileTxBytes()`, which return the traffic going over mobile data (also included in the total)
3. `getMobileRxPackets()` and `getMobileTxPackets()`, which are the packet counts for the mobile data connection

Per-Application Statistics

Technically, `TrafficStats` does not provide per-application traffic statistics. Rather, it provides per-UID traffic statistics. In most cases, the UID (user ID) of an application is unique, and therefore per-UID statistics map to per-application statistics. However, it is possible for multiple applications to share a single UID (e.g., via the `android:sharedUserId` manifest attribute) — in this case, `TrafficStats` would appear to provide traffic data for all applications sharing that UID.

There are per-UID equivalents of the first four methods listed in the previous section, replacing "Total" with "Uid". So, to find out overall traffic for an application, you could use `getUidRxBytes()` and `getUidTxBytes()`. However, these are the only two UID-specific methods that were implemented in API Level 8. Equivalents of the others (e.g., `getUidRxPackets()`) were added in API Level 12. API Level 12 also added some TCP-specific methods (e.g., `getUidTcpTxBytes()`). Note, though, that the mobile-only method are only available at the device level; there are no UID-specific versions of those methods.

Interpreting the Results

You will get one of two types of return value from these methods.

In theory, you will get the value the method calls for (e.g., number of bytes, number of packets). The documentation does not state the time period for that value, so while it is possible that it is really "number of bytes since the device was booted", we do not know that for certain. Hence, `TrafficStats` results should be used for comparison purposes, either comparing the same value over time or comparing multiple values at the same time. For example, to measure bandwidth consumption, you will need to record the `TrafficStats` values at one point in time, then again later — the difference between them represents the consumed bandwidth during that period of time.

In practice, while the "total" methods seem reliable, the per-UID methods may return `-1`. Three possible meanings are:

1. The device is old and is not set up to measure per-UID values
2. There has been no traffic of that type on that UID since boot, or
3. You do not have permission to know the traffic of that type on that UID

Hence, the per-UID values are a bit "hit or miss", which you will need to take into account.

Example: TrafficMonitor

To illustrate the use of `TrafficStats` methods and analysis, let us walk through the code associated with the `Bandwidth/TrafficMonitor` sample application. This is a simple activity that records a snapshot of the current traffic levels on startup, then again whenever you tap a button. On-screen, it will display the current value, previous value, and difference ("delta") between them. In Logcat, it will dump the same information on a per-UID basis.

TrafficRecord

It would have been nice if `TrafficStats` were indeed an object that you would instantiate, that captured the traffic values at that moment in time. Alas, that is not how it was written, so we need to do that ourselves. In the `TrafficMonitor` project, this job is delegated to a `TrafficRecord` class:

```
package com.commonsware.android.tuning.traffic;

import android.net.TrafficStats;

class TrafficRecord {
  long tx=0;
  long rx=0;
  String tag=null;

  TrafficRecord() {
    tx=TrafficStats.getTotalTxBytes();
    rx=TrafficStats.getTotalRxBytes();
  }

  TrafficRecord(int uid, String tag) {
    tx=TrafficStats.getUidTxBytes(uid);
    rx=TrafficStats.getUidRxBytes(uid);
```

```
    this.tag=tag;
  }
}
```

(from Bandwidth/TrafficMonitor/app/src/main/java/com/commonsware/android/tuning/traffic/TrafficRecord.java)

There are two separate constructors, one for the total case and one for the per-UID case. The total case just logs `getTotalRxBytes()` and `getTotalTxBytes()`, while the per-UID case uses `getUidRxBytes()` and `getUidTxBytes()`. The per-UID case also stores a "tag", which is simply a `String` identifying the UID for this record — as you will see, `TrafficMonitor` uses this for a package name.

TrafficSnapshot

An individual `TrafficRecord`, though, is insufficient to completely capture the traffic figures at a moment in time. We need a collection of `TrafficRecord` objects, one for the device ("total") and one per running UID. The work to collect all of that is handled by a `TrafficSnapshot` class:

```
package com.commonsware.android.tuning.traffic;

import java.util.HashMap;
import android.content.Context;
import android.content.pm.ApplicationInfo;

class TrafficSnapshot {
  TrafficRecord device=null;
  HashMap<Integer, TrafficRecord> apps=
    new HashMap<Integer, TrafficRecord>();

  TrafficSnapshot(Context ctxt) {
    device=new TrafficRecord();

    HashMap<Integer, String> appNames=new HashMap<Integer, String>();

    for (ApplicationInfo app :
          ctxt.getPackageManager().getInstalledApplications(0)) {
      appNames.put(app.uid, app.packageName);
    }

    for (Integer uid : appNames.keySet()) {
      apps.put(uid, new TrafficRecord(uid, appNames.get(uid)));
    }
  }
}
```

(from Bandwidth/TrafficMonitor/app/src/main/java/com/commonsware/android/tuning/traffic/TrafficSnapshot.java)

The constructor uses `PackageManager` to iterate over all installed applications and builds up a `HashMap`, mapping the UID to a `TrafficRecord` for that UID, tagged with the application package name (e.g., `com.commonsware.android.tuning.traffic`). It also creates one `TrafficRecord` for the device as a whole.

TrafficMonitorActivity

`TrafficMonitorActivity` is what creates and uses `TrafficSnapshot` objects. This is a fairly conventional activity with a TableLayout-based UI:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/table"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <Button
    android:onClick="takeSnapshot"
    android:text="Take Snapshot"/>

  <TableRow>

    <TextView
      android:layout_column="1"
      android:layout_gravity="right"
      android:text="@string/received"
      android:textSize="20sp"/>

    <TextView
      android:layout_gravity="right"
      android:text="@string/sent"
      android:textSize="20sp"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/latest"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
```

```
    android:id="@+id/latest_rx"
    android:layout_marginRight="@dimen/margin_right"
    android:gravity="right"
    android:textSize="20sp"/>

  <TextView
    android:id="@+id/latest_tx"
    android:gravity="right"
    android:textSize="20sp"/>
</TableRow>

<TableRow>

  <TextView
    android:layout_marginRight="@dimen/margin_right"
    android:gravity="right"
    android:text="@string/previous"
    android:textSize="20sp"
    android:textStyle="bold"/>

  <TextView
    android:id="@+id/previous_rx"
    android:layout_marginRight="@dimen/margin_right"
    android:gravity="right"
    android:textSize="20sp"/>

  <TextView
    android:id="@+id/previous_tx"
    android:gravity="right"
    android:textSize="20sp"/>
</TableRow>

<TableRow>

  <TextView
    android:layout_marginRight="@dimen/margin_right"
    android:gravity="right"
    android:text="@string/delta"
    android:textSize="20sp"
    android:textStyle="bold"/>

  <TextView
    android:id="@+id/delta_rx"
    android:layout_marginRight="@dimen/margin_right"
    android:gravity="right"
    android:textSize="20sp"/>

  <TextView
```

```
      android:id="@+id/delta_tx"
      android:gravity="right"
      android:textSize="20sp"/>
  </TableRow>

</TableLayout>
```

(from Bandwidth/TrafficMonitor/app/src/main/res/layout/main.xml)

The activity implementation consists of three methods. There is your typical `onCreate()` implementation, where we initialize the UI, get our hands on the `TextView` widgets for output, and take the initial snapshot:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    latest_rx=(TextView)findViewById(R.id.latest_rx);
    latest_tx=(TextView)findViewById(R.id.latest_tx);
    previous_rx=(TextView)findViewById(R.id.previous_rx);
    previous_tx=(TextView)findViewById(R.id.previous_tx);
    delta_rx=(TextView)findViewById(R.id.delta_rx);
    delta_tx=(TextView)findViewById(R.id.delta_tx);

    takeSnapshot(null);
  }
```

(from Bandwidth/TrafficMonitor/app/src/main/java/com/commonsware/android/tuning/traffic/TrafficMonitorActivity.java)

The `takeSnapshot()` method creates a new `TrafficSnapshot` (held in a `latest` data member) after moving the last `TrafficSnapshot` to a `previous` data member. It then updates the `TextView` widgets for the `latest` data and, if the `previous` data member is not `null`, also for the `previous` snapshot and the difference between them. This alone is sufficient to update the UI, but we also want to log per-UID data to Logcat:

```
  public void takeSnapshot(View v) {
    previous=latest;
    latest=new TrafficSnapshot(this);

    latest_rx.setText(String.valueOf(latest.device.rx));
    latest_tx.setText(String.valueOf(latest.device.tx));

    if (previous!=null) {
      previous_rx.setText(String.valueOf(previous.device.rx));
      previous_tx.setText(String.valueOf(previous.device.tx));
```

```
      delta_rx.setText(String.valueOf(latest.device.rx-previous.device.rx));
      delta_tx.setText(String.valueOf(latest.device.tx-previous.device.tx));
    }

    ArrayList<String> log=new ArrayList<String>();
    HashSet<Integer> intersection=new HashSet<Integer>(latest.apps.keySet());

    if (previous!=null) {
      intersection.retainAll(previous.apps.keySet());
    }

    for (Integer uid : intersection) {
      TrafficRecord latest_rec=latest.apps.get(uid);
      TrafficRecord previous_rec=
            (previous==null ? null : previous.apps.get(uid));

      emitLog(latest_rec.tag, latest_rec, previous_rec, log);
    }

    Collections.sort(log);

    for (String row : log) {
      Log.d("TrafficMonitor", row);
    }
  }
```

(from Bandwidth/TrafficMonitor/app/src/main/java/com/commonsware/android/tuning/traffic/TrafficMonitorActivity.java)

One possible problem with the snapshot system is that the process list may change between snapshots. One simple way to address this is to only log to Logcat data where the application's UID exists in both the `previous` and `latest` snapshots. Hence, `takeSnapshot()` uses a `HashSet` and `retainAll()` to determine which UIDs exist in both snapshots. For each of those, we call an `emitLog()` method to record the data to an `ArrayList`, which is then sorted and dumped to Logcat.

The `emitLog()` method builds up a line with the package name and bandwidth consumption information, assuming that there is bandwidth to report (i.e., we have a value other than `-1`):

```
  private void emitLog(CharSequence name, TrafficRecord latest_rec,
                       TrafficRecord previous_rec,
                       ArrayList<String> rows) {
    if (latest_rec.rx>-1 || latest_rec.tx>-1) {
      StringBuilder buf=new StringBuilder(name);

      buf.append("=");
```

```
      buf.append(String.valueOf(latest_rec.rx));
      buf.append(" received");

      if (previous_rec!=null) {
        buf.append(" (delta=");
        buf.append(String.valueOf(latest_rec.rx-previous_rec.rx));
        buf.append(")");
      }

      buf.append(", ");
      buf.append(String.valueOf(latest_rec.tx));
      buf.append(" sent");

      if (previous_rec!=null) {
        buf.append(" (delta=");
        buf.append(String.valueOf(latest_rec.tx-previous_rec.tx));
        buf.append(")");
      }

      rows.add(buf.toString());
    }
  }
```

(from Bandwidth/TrafficMonitor/app/src/main/java/com/commonsware/android/tuning/traffic/TrafficMonitorActivity.java)

Since the lines created by `emitLog()` start with the package name, and since we are sorting those before dumping them to Logcat, they appear in Logcat in sorted order by package name.

Using TrafficMonitor

Running the activity gives you the initial received and sent counts (in bytes):

Figure 1013: The TrafficMonitor sample application, as initially launched

Tapping Take Snapshot grabs a second snapshot and compares the two:

Figure 1014: The TrafficMonitor sample application, after Take Snapshot was clicked

Also, Logcat will show how much was used by various apps:

```
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.amblingbooks.bookplayerpro=880
received (delta=0), 3200 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.browser=19045241
received (delta=0), 2375847 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.downloads=27884469 received (delta=0), 9126 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.providers.telephony=2328
received (delta=0), 4912 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.vending=3271839 received
(delta=0), 260626 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.coair.mobile.android=887425
received (delta=0), 81366 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.commonsware.android.browser1=262553 received (delta=0), 7286 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.dropbox.android=6189833 received
(delta=0), 4298 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.evernote=3471398 received
(delta=0), 742178 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.genie.geniewidget=358816 received (delta=0), 17775 sent
(delta=0)
```

```
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.googlevoice=103255 received (delta=0), 35559 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.maps=28440829 received (delta=0), 1230867 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.backup=51320
received (delta=0), 49041 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.gm=10915084
received (delta=0), 14428803 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.googlequicksearchbox=37817 received (delta=0), 12554 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.syncadapters.contacts=1955990 received (delta=0), 714893 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.voicesearch=67948
received (delta=0), 121908 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.youtube=3128
received (delta=0), 2792 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.howcast.android.app=2250407
received (delta=0), 26727 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.rememberthemilk.MobileRTM=6836605 received (delta=0), 2902904 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.tripit=109499 received
(delta=0), 50060 sent (delta=0)
```

Other Ways to Employ TrafficStats

Of course, there are more ways you could use `TrafficStats` than simply having an activity to report them on a button click. `TrafficMonitor` is merely a demonstration of using the class and providing a lightweight way to get value out of that data. Depending upon your application's operations, though, you may wish to consider using `TrafficStats` in other ways, in your production code or in your test suites.

In Production

If your app is a bandwidth monitor, the need to use `TrafficStats` is obvious. However, even if your app does something else, you may wish to use `TrafficStats` to understand what is going on in terms of Internet access within your app or on the device as a whole.

For example, you might want to consider bandwidth consumption to be a metric worthy of including in the rest of the "analytics" you generate from your app. If you are using services like Flurry to monitor which activities get used and so on, you might consider also logging the amount of bandwidth your application consumes.

This not only gives you much more "real world" data than you will be able to collect on your own, but it may give you ideas of how users are using your application beyond what the rest of your metrics are reporting.

Another possibility would be to include your app's bandwidth consumption in error logs reported via libraries like ACRA. Just as device particulars can help identify certain bug report patterns, perhaps certain crashes of your app only occur when users are using a lot of bandwidth in your app, or using a lot of bandwidth elsewhere and perhaps choking your own app's Internet access.

The chapter on bandwidth mitigation strategies will also cover a number of uses of `TrafficStats` for real-time adjustment of your application logic.

During Testing

You might consider adding `TrafficStats`-based bandwidth logging for your application in your test suites. While individual tests may or may not give you useful data, you may be able to draw trendlines over time to see if you are consuming more or less bandwidth than you used to. Take care to factor in that you may have changed the tests, in addition to changing the code that is being tested.

From a JUnit-based unit test suite, measuring bandwidth consumption is not especially hard. You can bake it into the `setUp()` and `tearDown()` methods of your test cases, either via inheritance or composition, and log the output to a file or Logcat.

From an external test engine, like monkeyrunner or NativeDriver, recording bandwidth usage is more tricky, because your test code is not running on the device or emulator. You may have to include a `BroadcastReceiver` in your production code that will log bandwidth usage and trigger that code via the `am broadcast` shell command.

Measuring Bandwidth Consumption

The first step towards addressing bandwidth concerns is to get a better picture of how much bandwidth you are actually consuming, when, and under what conditions. Only then will you be able to determine where your efforts need to be applied and whether those efforts are actually giving you positive results. This chapter will examine a handful of ways you can determine how much bandwidth you are really using in your application.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

On-Device Measurement

Many times, you are best served by measuring your bandwidth consumption right on the device itself:

1. This is your only option for gathering bandwidth metrics from copies of your app in end users' hands, unless they invite you to their home or office and have you sniff on their personal network, which seems unlikely
2. This is your only option for gathering bandwidth metrics when you are using mobile data plans (e.g., 3G) instead of WiFi, since you probably do not control the wireless telecommunications infrastructure in your area
3. This is your simplest option for tying bandwidth metrics to events within your app or occurring on the device

4. This is your only option for using bandwidth metrics to adjust your application behavior in real time, in addition to using the metrics to learn how best to adjust your code in future updates to the app

Hence, in addition to perhaps other off-device techniques, you really should consider one of the on-device approaches outlined in the following sections.

Yourself, via TrafficStats

The preceding chapter outlined how to use the `TrafficStats` class to collect metrics on the bandwidth consumed by applications (including yours) and for the device as a whole. This gives you the most flexibility, because you can write your own code to collect whatever portion of this data you need. It can address all of the bullets shown above, for example.

It is not perfect, though:

1. It requires you to write your own code, adding yet more work to your plate
2. Per-UID traffic data may or may not be available, depending upon the device

Data Usage Screen in Settings

For more casual use, the Settings app in most Android devices offers a “Data Usage” screen that shows how much bandwidth has been consumed over a period of time:

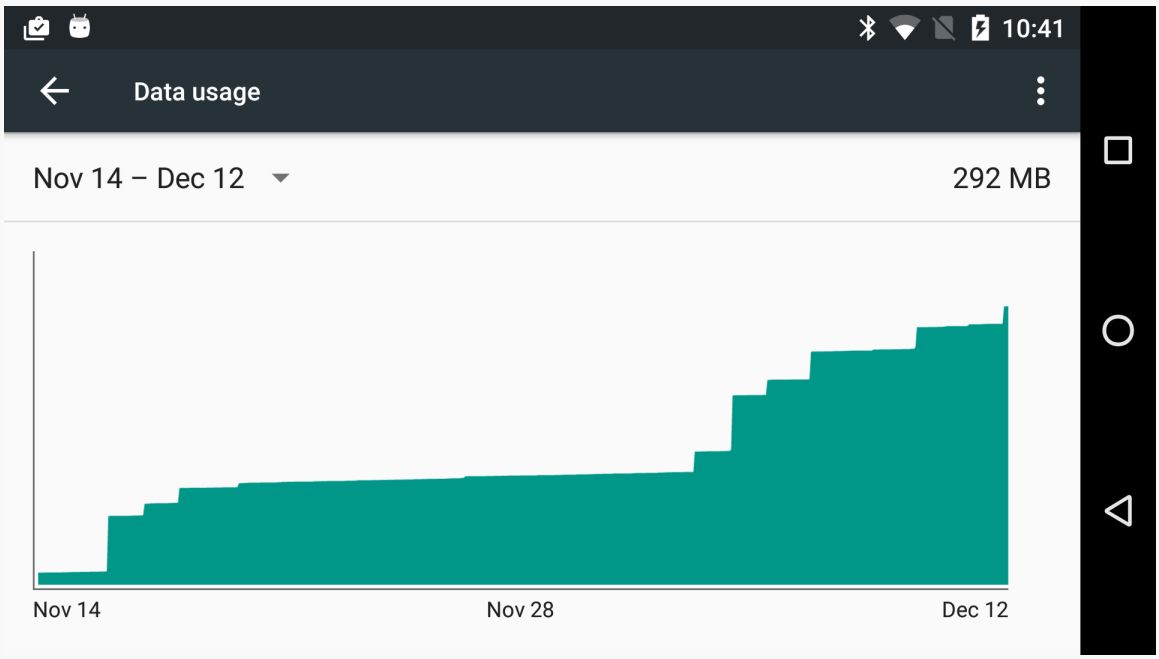


Figure 1015: Settings, App, Data Usage Screen, Data Usage Graph

Scrolling further down will give you details of what apps were involved in that data usage:

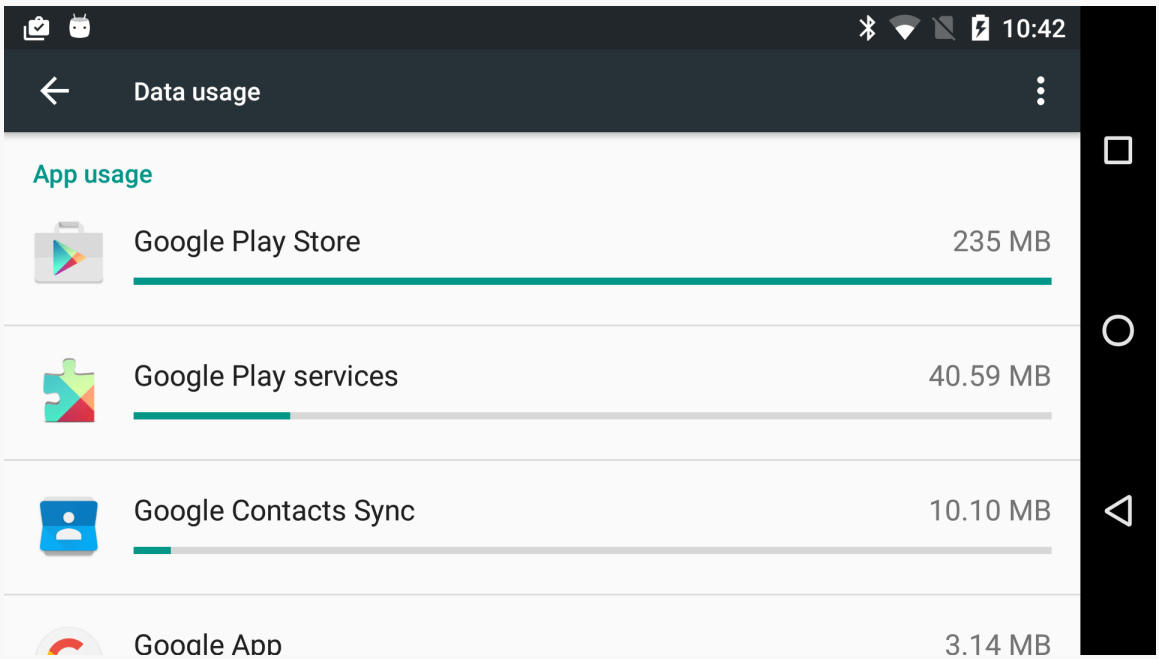


Figure 1016: Settings, App, Data Usage Screen, Data Usage "Blame List"

Tapping on any one of those list items will give you a bit more detail, specifically how much of that bandwidth was consumed while the app was in the foreground or the background:

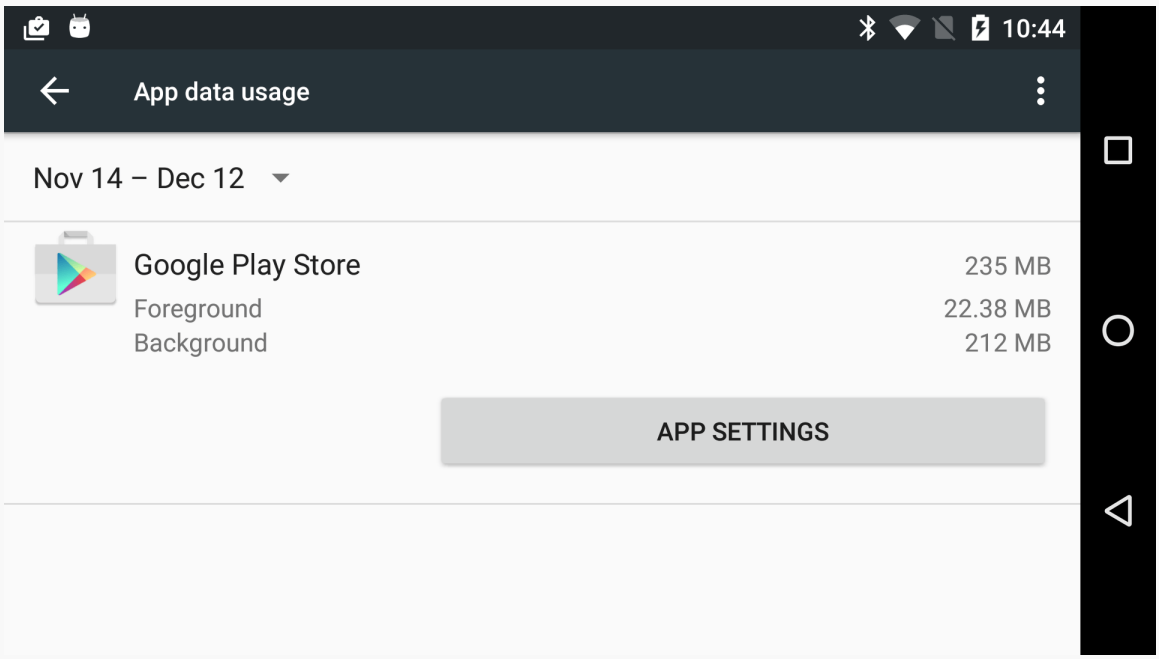


Figure 1017: Settings, App, Data Usage Screen, Data Usage App Details

Off-Device Measurement

The biggest limitation of `TrafficStats` is that it only gives you gross metrics: numbers of bytes, packets, and so on. Sometimes, that is not enough to help you understand why those bytes, packets, and so on are actually being sent or received. Sometimes, it would be nice to understand the traffic in more detail, from the ports and IP addresses to perhaps the actual data being transmitted. For obvious security reasons, this is not something an ordinary Android SDK application can do. However, there are techniques for accomplishing this, mostly for use over WiFi in your own home or office network. Some of these are outlined in the following sections.

Wireshark

Wireshark, formerly known as Ethereal, is perhaps the world's leading open source network traffic analyzer and packet inspector. Using it, you can learn in great detail what is going on with your local network. And, Android provides additional options for you to leverage Wireshark to make sense of application behavior. Wireshark is available for Linux, macOS, and Windows.

There is a lightly-documented `-tcpdump` switch available on the Android emulator. If you launch the emulator from the command line with that switch (plus `-avd` to identify the AVD file you want to use), all network access is dumped to your specified log file. You can then load that data into Wireshark for analysis, via File|Open from the main menu.

For example, here is a screenshot of Wireshark examining data from such an emulator dump file, in which the emulator was used to conduct a Google search:

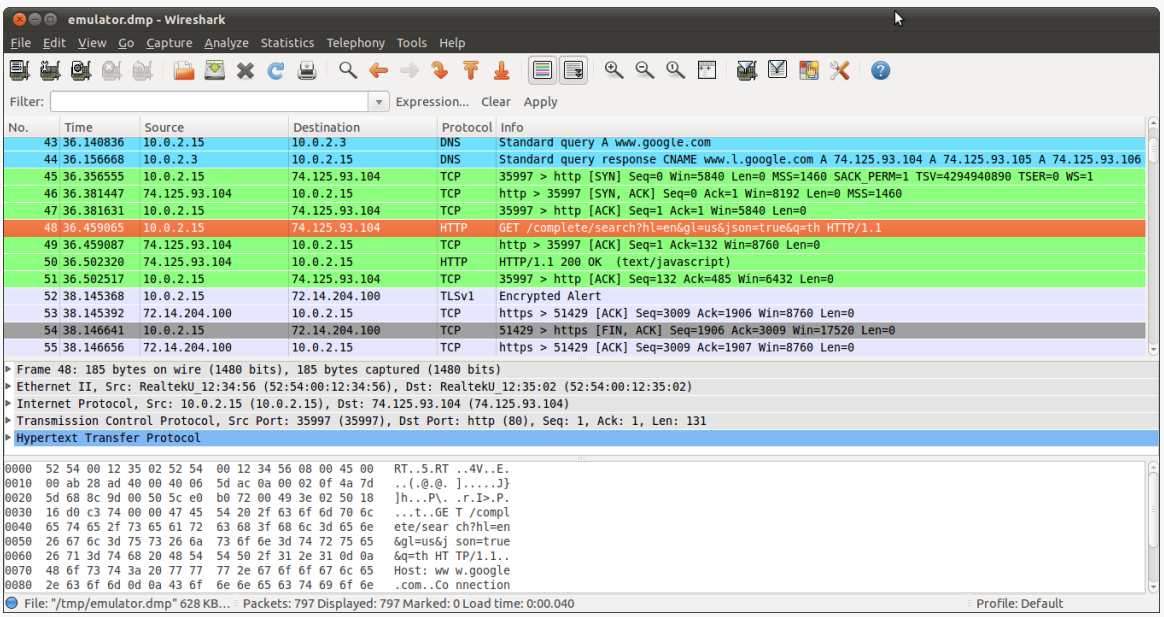


Figure 1018: Wireshark examining captured emulator packets

This screenshot shows an HTTP request in the highlighted line in the list, with the hex and ASCII contents of the request shown in the bottom pane.

In terms of using Wireshark to monitor traffic from actual hardware, that is indubitably possible. However, WiFi packet collection is a tricky process with Wireshark, being very dependent upon operating system and possibly even the WiFi adapter chipset. You also get much lower-level information, making it a bit more challenging to figure out what is going on. Attempting to cover all of this is well beyond the scope of this book and the author's Wireshark expertise.

Networking Hardware

Sophisticated firewalls sometimes have packet tracing/sniffing capability. In this case, "sophisticated" does not necessarily mean "expensive", as open source router/firewall distributions, like OpenWrt, can be used for this sort of work. In this case, the router captures the packets and, in many cases, routes them to Wireshark for analysis. Some might offer on-board analysis (e.g., Web interface to packet capture logs).

This is particularly useful on a Windows wireless network. Wireshark has limits, imposed by Windows, that cause some problems when trying to capture WiFi packets. By offloading the packet capture to networking hardware, those limits can be bypassed.

Android Studio Profiler

`TrafficStats` is great for measuring gross bandwidth consumption over some period of time. However, it requires coding, logging, and your own analysis mechanism.

In Android Studio, the Android Profiler tool allows you to examine the real-time behavior of your app with respect to various system resources, such as bandwidth consumption:

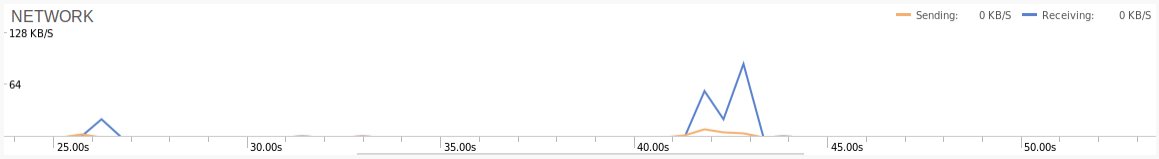


Figure 1019: Android Studio, Android Profiler, Network Graph

Clicking on the graph gives you a bit more detail, including the state of various radios and the number of connections that were in use:

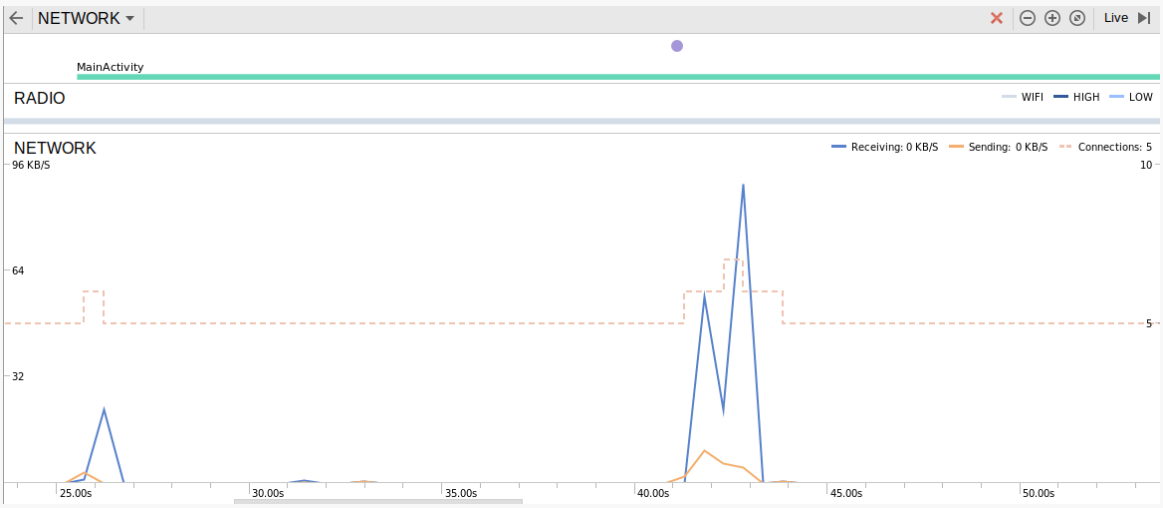


Figure 1020: Android Studio, Android Profiler, Detailed Network Graph

This graph comes from a Picasso demo application from earlier in the book, which retrieves the latest 25 `android` questions on Stack Overflow and shows them in a `ListView`, along with the avatar for the person asking the question. The graph shows the initial load of data from the Stack Exchange JSON API, followed ~15 seconds later by some scrolling in the app, forcing Picasso to go load more avatars.

As this responds in near-real-time to things you do in your app, you can see if the graph shows network accesses at unexpected times, or more bandwidth consumed than you might expect. For example, once all the avatars were loaded, no more bandwidth should be consumed by the Picasso sample app, assuming all the avatars could fit in Picasso's memory cache.

Being Smarter About Bandwidth

Given that you are collecting metrics about bandwidth consumption, you can now start to determine ways to reduce that consumption. You may be able to permanently reduce that consumption (at least on a per-operation basis). You may be able to shunt that consumption to times or networks that the user prefers. This chapter reviews a variety of means of accomplishing these ends.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Internet access.

Bandwidth Savings

The best way to reduce bandwidth consumption is to consume less bandwidth.

(in other breaking news, water is wet)

In recent years, developers have been able to be relatively profligate in their use of bandwidth, pretty much assuming everyone has an unlimited high-speed Internet connection to their desktop or notebook and the desktop or Web apps in use on them. However, those of us who lived through the early days of the Internet remember far too well the challenges that dial-up modem accounts would present to users (and perhaps ourselves). Even today, as Web apps try to "scale to the Moon and back", bandwidth savings becomes important not so much for the end user, but for the Web app host, so its own bandwidth is not swamped as its user base grows.

Fortunately, widespread development problems tend to bring rise to a variety of solutions — a variant on the "many eyes make bugs shallow" collaborative development phenomenon. Hence, there are any number of tried-and-true techniques for reducing bandwidth consumption that have had use in Web apps and elsewhere. Many of these are valid for native Android apps as well, and a few of them are profiled in the following sections.

Classic HTTP Solutions

Trying to get lots of data to fit on a narrow pipe — whether that pipe is on the user's end or the provider's end — has long been a struggle in Web development. Fortunately, there are a number of ways you can leverage HTTP intelligently to reduce your bandwidth consumption.

GZip Encoding

By default, HTTP requests and response are uncompressed. However, you can enable GZip encoding and thereby request that the server compress its response, which is then decompressed on the client. This trades off CPU for bandwidth savings and therefore needs to be done judiciously.

Enabling GZip compression is a two-step process:

- Adding the `Accept-Encoding: gzip` header to the HTTP request
- Determine if the response was compressed and, if so, decompressing it

Bear in mind that the Web server may or may not honor your GZip request, for whatever reason (e.g., response is too small to make it worthwhile).

If-Modified-Since / If-None-Match

Of course, avoiding a download offers near-100% compression. If you are caching data, you can take advantage of HTTP headers to try to skip downloads that are the same content as what you already have, specifically `If-Modified-Since` and `If-None-Match`.

An HTTP response can contain either a `Last-Modified` header or an `ETag` header. The former will contain a timestamp and the latter will contain some opaque value. You can store this information with the cached copy of the data (e.g., in a database table). Later on, when you want to ensure you have the latest version of that file,

your HTTP GET request can include an `If-Modified-Since` header (with the cached Last-Modified value) or an `If-None-Match` header (with the cached `ETag` value). In either case, the server should return either a `304` response, indicating that your cached copy is up to date, or a `200` response with the updated data. As a result, you avoid the download entirely (other than HTTP headers) when you do not need the updated data.

Binary Payloads

While XML and JSON are relatively easy for humans to read, that very characteristic means they tend to be bloated in terms of bandwidth consumption. There are a variety of tools, such as Google's Protocol Buffers and Apache's Thrift, that allow you to create and parse binary data structures in a cross-platform fashion. These might allow you to transfer the same data that you would in XML or JSON in less space. As a side benefit, parsing the binary responses is likely to be faster than parsing XML or JSON. Both of these tools involve the creation of an IDL-type file to describe the data structure, then offer code generators to create Java classes (or equivalents for other languages) that can read and write such structures, converting them into platform-neutral on-the-wire byte arrays as needed.

Minification

If you are loading JavaScript or CSS into a `WebView`, you should consider standard tricks for compressing those scripts, collectively referred to as "minification". These techniques eliminate all unnecessary whitespace and such from the files, rename variables to be short, and otherwise create a syntactically-identical script that takes up a fraction of the space.

Keep-Alive Semantics

A chunk of the overhead involved in HTTP operations is simply establishing the socket connection with the Web server. Advertising that you want the socket to be kept alive, in anticipation of upcoming follow-on requests, can reduce this overhead.

Using higher-level HTTP clients, like OkHttp, helps here, because usually they handle all the details of keeping the socket open.

With SSL, though, keep-alive was not an option, until Google released the SPDY specification. SPDY in turn formed the basis of HTTP/2, the new standard for Web

communications (replacing the venerable HTTP/1.1). OkHttp supports SPDY and HTTP/2.

Push versus Poll

Another way to consume less bandwidth is to only make the requests when it is needed. For example, if you are writing an email client, the way to use the least bandwidth is to download new messages only when they exist, rather than frequently polling for messages.

Off the cuff, this may seem counter-intuitive. After all, how can we know whether or not there are any messages if we are not polling for them?

The answer is to use a low-bandwidth push mechanism. The quintessential example of this is GCM, the Google Cloud Messaging system, available for Android 2.2 and newer. This service from Google allows your application to subscribe to push notifications sent out by your server. Those notifications are delivered asynchronously to the device by way of Google's own servers, using a long-lived socket connection. All you do is register a `BroadcastReceiver` to receive the notifications and do something with them.

For example, Remember the Milk — a task management Web site and set of mobile apps — uses GCM to alert the device of task changes you make through the Web site. Rather than the Remember the Milk app having to constantly poll to see if tasks were added, changed, or deleted, the app simply waits for GCM events.

You could create your own push mechanism, perhaps using a WebSocket or MQTT. The downside is that you will need a service in memory all of the time to manage the socket and thread that monitors it. If you only need this while your service is in memory for other reasons, that is fine. However, keeping a service in memory 24x7 has its own set of issues, not the least of which is that users will tend to smack it down using a "task killer" or the Manage Services screen in the Settings app. Doze mode on Android 6.0+ will also cause problems with this approach.

Thumbnails and Tiles

A general rule of thumb is: don't download it until you really need it.

Sometimes, you do not know if you really need a particular item until something happens in the UI. Take a `ListView` displaying thumbnails of album covers for a music app. Assuming the album covers are not stored locally, you will need to

download them for display. However, which covers you need varies based upon scrolling. Downloading a high-resolution album cover that might get tossed in a matter of milliseconds (after an expensive rescale to fit a thumbnail-sized space) is a waste of bandwidth.

In this case, either the album covers are something you control on the server side, or they are not. If they are, you can have the server prepare thumbnails of the covers, stored at a spot that the app can know about (e.g., `.../cover.jpg` it is `.../thumbnail.jpg`). The app can then download thumbnails on the fly and only grab the full-resolution cover if needed (e.g., user clicks on the album to bring up a detail screen). If you do not control the album covers, this option might still be available to you if you can run your own server for the purposes of generating such thumbnails.

You can see a similar effect with the map tiles in Google Maps. When zooming out, the existing map tiles are scaled down, with placeholders (the gridlines) for the remaining spots, until the tiles for those spots are downloaded. When zooming in, the existing map tiles are scaled up with a slight blurring effect, to give the user some immediate feedback while the full set of more-detailed tiles is downloaded. And, if the user pans, you once again get placeholders while the tiles for the newly uncovered areas are downloaded. In this fashion, Google Maps is able to minimize bandwidth consumption by giving users partial results immediately and back-filling in the final results only when needed. This same sort of approach may be useful with your own imagery.

Bandwidth Shaping

Sometimes, you have no ability to reduce the bandwidth itself. Perhaps you do not control both ends of the communications pipeline. Perhaps the data you are trying to exchange is already compressed (e.g., downloading an MP4 video). Perhaps some of the techniques in the preceding section were unavailable to you (e.g., cannot route data through third-party servers like Google's for GCM).

There still may be ways for you to help your users, by shaping your bandwidth use. Rather than just blindly doing whatever you want whenever you want, you learn what the *user* wants and what *other applications* want and tailor your bandwidth use on the fly to match those needs. The following sections outline some ways of achieving this.

Driven by Preferences

If you are consuming enough bandwidth that this chapter is relevant to you, you probably are consuming enough bandwidth that you should be asking the user how best to consume that bandwidth. After all, they are the one paying the price — in time as well as money – for that consumption.

The following sections present some possible strategies for preference-based bandwidth shaping.

Budgets

One strategy is for the user to give you a budget (e.g., 20MB/day) and for you to stick within that budget.

Collecting the budget is fairly easy — just use `SharedPreferences`. Either use a `ListPreference` with likely budget value or an `EditTextPreference` and a bit of validation for a free-form budget amount.

Next, you will need to have some idea how much bandwidth any given network operation will consume. For some things, this might be an estimate based on your experiments as a developer, or perhaps it is based on historical averages for this user and type of operation. For example, a "podcatcher" (feed reader designed to download podcast episodes) should have some idea how big a given RSS or Atom feed download should be. In some cases, it might be worthwhile to get a better estimate — for example, the podcatcher might use an HTTP `HEAD` request to determine the size of the MP3 or OGG file before deciding whether to download it.

Then, you need to be keeping track of your budget. This could be a simple flat file with the initial `TrafficStats` bandwidth values for your process. Re-initialize that file on the first network operation of the day (or whatever period you chose for your budget). Before doing another network operation, compare the current `TrafficStats` values with the initial ones and see how close you are to the budget. If the new network operation will exceed the budget, skip the operation, perhaps putting it in a work queue to perform in the next budget. You might even hold a reserve for certain types of operations. For example, the podcatcher might ensure there is at least 10% of the budget available for downloading the feeds, even if it means putting a podcast on the queue for download tomorrow. That way, you can present to the user the latest podcast information, with icons indicating which are

downloaded and which are queued for download — the user might be able to then request to override the budget and download something on demand.

For devices that lack per-UID `TrafficStats` support, you will have to "fake it" a bit. Use your own calculations of how much bandwidth each operation consumes and track that information, even if you wind up missing out on some bytes here or there.

Connectivity

If the user might not care how much bandwidth you consume, so long as it is unmetered bandwidth, you might include a `CheckBoxPreference` to indicate if large network operations should be limited to WiFi and avoid mobile data.

You could then use `ConnectivityManager` and `getActiveNetworkInfo()` to see what connection you have before performing a network operation. If it is a background operation (e.g., the podcatcher checking for new podcasts every hour), if the network is not the desired one, you can skip the operation or put it on a work queue for re-trying later. If it is a foreground operation (e.g., the user clicked a "refresh" menu choice), you could pop up a confirmation `AlertDialog` to warn the user that they are on mobile data — perhaps this time they are interested in doing the operation anyway.

Another approach for handling the background operations is to register a `BroadcastReceiver` for the `CONNECTIVITY_ACTION` broadcast (defined on `ConnectivityManager`). If the connectivity switches to mobile data, cancel your outstanding `AlarmManager` alarms; if connectivity switches to WiFi, re-enable those alarms.

Of course, you should also consider monitoring the background data setting — the global Settings checkbox indicating whether background network operations are allowed. On `ConnectivityManager`, `getBackgroundDataSetting()` tells you the state of this checkbox, and `ACTION_BACKGROUND_DATA_SETTING_CHANGED` allows you to set up a `BroadcastReceiver` to watch for changes in its state.

Windows

If your user is less concerned about the bandwidth or the network, but does care about the time of day (e.g., does not want your application consuming significant bandwidth when they might be getting a VOIP call), you could offer preferences for that as well. Cook up a `TimePreference` and use that to collect start and stop times

for the high-bandwidth window. Then, set up alarms with `AlarmManager` for those points in time. The alarm for the start time of the window sets up a third alarm with your regular polling interval. The alarm for the stop time of the window cancels the polling interval alarm.

Driven by Other Usage

If your network I/O is part of a foreground application, one presumes that you are the most important thing in the user's life right now. Or, at least, the most important thing on the user's phone right now. Hence, what other applications might want to do with the Internet connection is not a major concern.

If, however, your network I/O is part of a background operation, it might be nice to try to avoid doing things that might upset the user. If the user is watching streaming video or is on a VOIP call or otherwise is aware of bandwidth changes, the bandwidth you use might impact the user in ways that the user will not appreciate very much. This is unlikely to be a big problem for small operations (e.g., downloading a 1KB JSON file), but larger operations (e.g., downloading a 5MB podcast) might be more noticeable.

You can use `TrafficStats` to help here. Before doing the actual network I/O, grab the current traffic data, wait a couple of seconds, and compare the latest to the previous values. If little to no bandwidth was consumed during that period, assume it is safe and go ahead and do your work. If, however, a bunch of bandwidth was consumed, you might want to consider:

1. Skipping this polling cycle and trying again later, or
2. Adding a one-off alarm using set() on `AlarmManager` to give you control again in a minute, with the current traffic data packaged as an extra on the `Intent`, so you can make a decision after a bigger sample size of bandwidth consumption, or
3. Adding an entry in a persistent work queue, so you know later on to try again if bandwidth contention has improved

You could try to get more sophisticated, by using `ActivityManager` and the per-UID values from `TrafficStats` to see if it is a foreground application that is the one consuming the bandwidth. It is unclear how reliable this will be, both in determining who is consuming the bandwidth (again, per-UID traffic is not available on many devices) and in avoid user angst. It may be simpler just to assume the worst and side-step your I/O until the other apps have quieted down.

Avoiding Metered Connections

Android 4.1 added `isActiveNetworkMetered()` as a method on `ConnectivityManager`. In principle, this will return `true` if Android thinks that the current data connection may involve bandwidth charges. You can examine this value and steer your bandwidth consumption accordingly.

Android 5.0 added `JobScheduler`, as an alternative to `AlarmManager` for arranging periodic work. One feature of `JobScheduler` is that you can indicate that certain jobs require Internet access, in which case Android will not bother giving you control unless such access is available. A further refinement is that you can state that a job requires an unmetered Internet connection, so you avoid doing bandwidth-hogging work on an expensive connection.

Data Saver

Android has had a per-app "data saver" mode for some time, with an eye towards reducing bandwidth consumption when the device is using a known metered data plan. Android 7.0 extends this to a device-wide setting,

Apps can be in one of three states as a result:

- The device is normal
- The device is in data-saver mode
- The device is in data-saver mode, but your app is whitelisted by the user

The idea is that if the device is in normal mode, you can do what you want. If the device is in data-saver mode, you should restrict your bandwidth, even if the user whitelists you. Apps that are not whitelisted have no network access while in the background.

To that end, `ConnectivityManager` has three things for you.

First, `isActiveNetworkMetered()` will return `true` if the device is on a metered data connection, `false` otherwise. This has been around for years (API Level 16+), but has not been all that popular, apparently.

Second, Android 7.0 has a `getRestrictBackgroundStatus()` method on `ConnectivityManager`. This returns an `int` that resolves to one of three values:

- `RESTRICT_BACKGROUND_STATUS_DISABLED`
- `RESTRICT_BACKGROUND_STATUS_ENABLED`
- `RESTRICT_BACKGROUND_STATUS_WHITELISTED`

If `isActiveNetworkMetered()` is `true`, and `getRestrictBackgroundStatus()` returns `RESTRICT_BACKGROUND_STATUS_ENABLED`, any attempts to use the network may fail, and so your app should plan accordingly.

If you want to try to react in real-time to changes in the data-saver configuration, you can register a receiver for `ACTION_RESTRICT_BACKGROUND_CHANGED` (defined on `ConnectivityManager`). This will be broadcast for any change in data-saver settings, which means that *your* app's state may not have changed. You will need to call `getRestrictBackgroundStatus()` to find out your current state. Also note that this broadcast is only sent to receivers registered dynamically, via `registerReceiver()`. You cannot register for this broadcast in the manifest.

To try to get on the whitelist, you might be tempted to try using `ACTION_IGNORE_BACKGROUND_DATA_RESTRICTIONS_SETTINGS` to lead the user to add your app to the Data Saver whitelist, so you have normal background network access. However, bear in mind that Google has a similar feature for the battery saver whitelist... and trying to use *that* action got apps banned from the Play Store. At the moment, there is no similar language around the use of the data saver whitelist... but, then again, they did not tell you they were going to ban you for asking to be on the battery saver whitelist until after Android 6.0 shipped.

Issues with Application Heap

RAM. Developers nowadays are used to having lots of it, and a virtual machine capable of using as much of it as exists (and more, given swap files and page files).

"Graybeards" — like the author of this book — distinctly remember a time when we had 16KB of RAM and were happy for it. Such graybeards would also appreciate it if you would get off their respective lawns.

Android comes somewhere in the middle. We have orders of magnitude more RAM than, say, the TRS-80 Model III. We do not have as much RAM as does the modern notebook, let alone a Web server. As such, it is easy to run out of RAM if you do not take sufficient care.

There are two facets of memory issues with Android:

- What are the problems we encounter inside our own app, in terms of our application heap?
- What problems can we encounter with system RAM overall, and how can we resolve them?

This part of the book examines memory-related issues, with this chapter focusing on the application heap. Another chapter will deal with system RAM issues. These are not to be confused with any memory-related issues inherent to graybeards.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Android's process model.

You Are in a Heap of Trouble

When we think of "memory" and Java-style programming, the primary form of memory is the heap. The heap holds all of our Java objects – from an `Activity` to a widget to a `String`.

Traditional Java applications have an initial heap size determined by the virtual machine, possibly configured via command-line options when the program was run. Traditional Java applications can also request additional memory from the OS, up to some maximum, also configurable.

Android applications have the same basic structure, with very limited configurability and much lower maximums than you might expect.

The original Android devices had a heap limit of 16MB. As screens increase in resolution, the heap limit tends to rise, but only to a point. 32MB to 64MB of heap space is fairly typical, but less-expensive devices, such as Android One models, will tend towards the lower end of that range.

This heap limit can be problematic. For example, each widget or layout manager instance takes around 1KB of heap space. This is why `AdapterView` provides the hooks for view recycling — we cannot have a `ListView` with literally thousands of row views without potentially running out of heap.

API Level 11+ supports applications requesting a "large heap". This is for applications that specifically need tons of RAM, such as an image editor to be used on a tablet. This is not for applications that run out of heap due to leaks or sloppy programming. Bear in mind that users will feel effects from large-heap applications, in that their other applications will be kicked out of memory more quickly, possibly irritating them. Also, garbage collection on large-heap applications runs more slowly, consuming more CPU time. To enable the large heap, add `android:largeHeap="true"` to the `<application>` element of your manifest. Finally, bear in mind that your "large heap" may not be any bigger than your regular heap would have been, as the "large heap" size is determined by the device manufacturer and takes into account things like available system RAM.

Determining Your Heap Size At Runtime

To get a sense for how much heap you will be able to potentially grow to, you can call `getMemoryClass()` on an `ActivityManager`. This will return your per-process heap limit in megabytes.

If you requested `android:largeHeap="true"` in the manifest, use `getLargeMemoryClass()` on `ActivityManager` to learn how large your "large heap" actually is. Note that it is entirely possible that the "large heap" is not all that large, or potentially is no bigger than the standard heap, depending upon how much RAM is physically present on the device.

Fragments of Memory

The Dalvik garbage collector is a non-compacting implementation, which makes `OutOfMemoryError` messages somewhat more likely than you would find on traditional Java environments.

Here, "non-compacting" means that Dalvik does not try to move objects around in physical memory to "compact" the use of physical memory, leaving a large contiguous block of free physical memory for future allocations.

For example, suppose that we allocate three 1K byte arrays, named A, B, and C. As it turns out, they were allocated using adjacent portions of physical memory, so that the last byte of A immediately precedes the first byte of B, and so on. Hence, we consumed 3K of available heap space to create these three 1K blocks.

If we release all references to A and B, they can be garbage-collected. Dalvik, like Java, will see that A and B are adjacent and will free up their physical memory, such that the memory is available as one contiguous 2K block for future allocations.

If, however, we release all references to A and C instead of A and B, Dalvik would be unable to make their blocks be contiguous, and so our heap would have two free 1K blocks, in addition to whatever other free memory that the heap already had.

Hence, allocating memory not only ties up that memory while it is in use, but it may *fragment* the memory even when it is released, such that our formerly pristine heap is now comprised of lots of little free blocks of space, separated from other such blocks by in-use objects. When we try to make a large allocation, such as setting up a byte array for a large image, it may be that while we have enough *total* heap

available for the request, there is no single block that would meet our request, and so we get an `OutOfMemoryError`.

One technique to help address this is to pre-allocate any large buffers that you know you need, up front when your process starts up, such as via a custom `Application` subclass. Then, use an "object pool" approach to obtain, use, and reuse these pre-allocated buffers, rather than having them be garbage-collected and have to be re-allocated later.

ART — the runtime engine used on Android 5.0+ — has a compacting garbage collector. However, it only compacts the heap when the app is in the background. So long as your application is in the foreground, ART behaves like Dalvik does, and your heap will continue to fragment.

Getting a Trim

It would be nice if we knew when a good time would be to cut back on our heap usage. For example, if we are caching a lot of data in our process, to save on future disk I/O, we could free up those caches at some point to help minimize our heap usage.

Fortunately, Android has some hooks for doing just that.

onTrimMemory() Callbacks

Starting in API Level 14, your activities, services, content providers, and custom `Application` classes all offer an `onTrimMemory()` method that you can override. This will be called from time to time to let you know about changes in the state of your app that might indicate it is time to free up some caches or otherwise cut back on memory consumption.

`onTrimMemory()` is passed a "level", indicating how serious the memory crunch is. At the present time, there are seven such levels, but others may be added in future versions of Android. However, these levels are in priority order, and the documentation indicates that Google will ensure that future levels are slotted into the order as appropriate. Hence, you can watch for levels of a certain severity *or higher* and take appropriate action at those points in time.

The seven levels are all defined as constants on the `ComponentCallbacks2` interface that defines `onTrimMemory()`. Four were defined in API Level 14, while the remaining three were defined in API Level 16. They are (in order of increasing severity):

- `TRIM_MEMORY_RUNNING_MODERATE` (added in API Level 16)
- `TRIM_MEMORY_RUNNING_LOW` (added in API Level 16)
- `TRIM_MEMORY_RUNNING_CRITICAL` (added in API Level 16)
- `TRIM_MEMORY_UI_HIDDEN` (added in API Level 14)
- `TRIM_MEMORY_BACKGROUND` (added in API Level 14)
- `TRIM_MEMORY_MODERATE` (added in API Level 14)
- `TRIM_MEMORY_COMPLETE` (added in API Level 14)

In particular, `TRIM_MEMORY_BACKGROUND` (or higher) indicates that your process is now on the list of processes to terminate to free up memory, and so the more memory you can free up, the less likely it is that your process will be terminated. Also, at `TRIM_MEMORY_UI_HIDDEN` or higher, your UI is no longer visible to the user, and so this is a fine time to free up UI-related memory that is safe to release, such as perhaps widget hierarchies that you would be rebuilding in `onResume()` later on anyway.

Note that while the focus tends to be on activities implementing `onTrimMemory()` to clean up UI-related resources, you are welcome to implement `onTrimMemory()` in services, content providers, and any custom `Application` subclass, so that you can free up memory that those may be managing as caches.

In the chapter on system RAM, we will get into why freeing up memory may help keep your process around, as we discuss the relationship between your application heap and available system RAM.

Warning: Contains Graphic Images

However, the most likely culprit for `OutOfMemoryError` messages are bitmaps. Bitmaps take up a remarkable amount of heap space. Developers often look at the size of a JPEG file and think that “oh, well, that’s only a handful of KB”, without taking into account:

1. the fact that most image formats, like JPEG and PNG, are compressed, and Android needs the uncompressed image to know what to draw
2. the fact that each pixel may take up several bytes (2 bytes per pixel for `RGB_565`, 3 bytes per pixel for `RGB_888`)

3. what matters is the resolution of the bitmap in its original form, as much (if not more) than the size in which it will be rendered – an 800x480 image displayed in an 80x48 `ImageView` still consumes 800x480 worth of pixel data
4. there are an awful lot of pixels in an image — 800 times 480 is 384,000

Android can make some optimizations, such as only loading in one copy of a `Drawable` resource no matter how many times you render it. However, in general, each bitmap you load takes a decent sized chunk of your heap, and too many bitmaps means not enough heap. It is not unheard of for an application to have more than half of its heap space tied up in various bitmap images.

Compounding this problem is that bitmap memory, before Android 3.0, was difficult to measure. In the actual Dalvik heap, a `Bitmap` would need ~80 bytes or so, regardless of image size. The actual pixel data was held in "native heap", the space that a C/C++ program would obtain via calls to `malloc()`. While this space was still subtracted from the available heap space, many diagnostic programs — such as MAT, to be examined in the next chapter — will not know about it. Android 3.0 moved the pixel data into the Dalvik heap, which will improve our ability to find and deal with memory leaks or overuse of bitmaps.

Bitmap Caching

Many Android libraries, like Picasso, offer bitmap caching. Using an existing caching implementation is a lot easier than is rolling your own.

However:

- Make sure that the library is intelligently sizing the cache based upon possible heap space, such as via `getMemoryClass()` as is noted earlier in this chapter.
- Where possible, tie your cache to `onTrimMemory()` so that you can flush that cache when appropriate.
- Be careful about using *multiple* libraries, each of which might implement its own cache, that you do not wind up caching too much overall. Each library tends to think that it is The One True Cache for your app and will be oblivious to any other caches — bitmap or otherwise — that you may have in your app. Ideally, the library will have a way for you to set the maximum cache size.

Bitmap Sizing

Sometimes, you do not need a full-size image. For example, if you are showing thumbnails of images in a `ListView`, but only expect to show the full-size image for a few (e.g., rows that the user clicks upon), it is wasteful to load the full-size image for everything in the list.

`BitmapFactory.Options` offers `inSampleSize`, which tells the framework to sample the image as it is loaded, to result in a smaller image. `inSampleSize` of 2 will result in an image that is half the width and half the height; `inSampleSize` of 4 will result in an image that is a quarter the width and a quarter the height; etc. Note that `inSampleSize` is limited to powers of 2 and will round as needed.

If you know ahead of time the size of the image, you can calculate an appropriate `inSampleSize` to use. Otherwise, for local content, you can use `BitmapFactory` twice:

- Once with a `BitmapFactory.Options` set with `inJustDecodeBounds` set to `true`, which will merely tell you how big the image is via `outHeight` and `outWidth` on the `BitmapFactory.Options` itself
- Once with a `BitmapFactory.Options` set with `inSampleSize` set to your desired value and `inJustDecodeBounds` set to `false`, to really load the image, but downsampled to consume less memory

This approach does not work well for images being downloaded directly from the Internet, as you do not want to download the image twice, once just to figure out how big it is. Instead, download the image *without* using `BitmapFactory` to a local file, then use `BitmapFactory` to load in the image. If you are electing to use a two-level cache (memory plus disk), you might download the image to the disk cache, for example.

For example, let's look at the `Bitmaps/InSampleSize` sample project, which demonstrates the memory impact (and visual impact) of loading bitmaps at varying sample sizes.

In the `assets/` directory, we have a ~70KB JPEG file of a flower (courtesy of the Wikimedia Project) and a ~50KB PNG of the CommonsWare logo. Both images are 672 pixels square, which makes them relatively large images. These are in assets to ensure that Android will not attempt any sort of density-based conversion of the images, if they were to be in a drawable resource directory.

The `MainActivity` of the project simply loads up a `ViewPager` and attaches it to a `SampleAdapter`:

```
package com.commonsware.android.bitmap.iss;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);

    pager.setAdapter(buildAdapter());
  }

  private PagerAdapter buildAdapter() {
    return(new SampleAdapter(this, getFragmentManager()));
  }
}
```

(from Bitmaps/InSampleSize/app/src/main/java/com/commonsware/android/bitmap/iss/MainActivity.java)

`SampleAdapter`, in turn, populates the `ViewPager` with four instances of a `BitmaFragment`, where we supply the `newInstance()` factory method of `BitmapFragment` with a value of 1, 2, 4, or 8 (`1 << position`), indicating the `inSampleSize` value we want to use for that fragment instance:

```
package com.commonsware.android.bitmap.iss;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  Context ctxt=null;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }
```

```
  @Override
  public int getCount() {
    return(4);
  }

  @Override
  public Fragment getItem(int position) {
    return(BitmapFragment.newInstance(1 << position));
  }

  @Override
  public String getPageTitle(int position) {
    return(BitmapFragment.getTitle(ctxt, 1 << position));
  }
}
```

(from Bitmaps/InSampleSize/app/src/main/java/com/commonsware/android/bitmap/iss/SampleAdapter.java)

`BitmapFragment` then:

- Inflates a layout consisting of four `ImageView` widgets, two at `672dp` square for the "natural" size (scaling only for density), and two at `128dp` square to illustrate how the images appear when constrained to a smaller space:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/byte_count"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <ImageView
      android:id="@+id/flower_large"
      android:layout_width="672dp"
      android:layout_height="672dp"
```

```
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:contentDescription="@string/flower_large"
      android:scaleType="fitCenter"/>

    <ImageView
      android:id="@+id/logo_large"
      android:layout_width="672dp"
      android:layout_height="672dp"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:contentDescription="@string/logo_large"
      android:scaleType="fitCenter"/>

    <ImageView
      android:id="@+id/flower_small"
      android:layout_width="128dp"
      android:layout_height="128dp"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:contentDescription="@string/flower_small"
      android:scaleType="fitCenter"/>

    <ImageView
      android:id="@+id/logo_small"
      android:layout_width="128dp"
      android:layout_height="128dp"
      android:layout_gravity="center_horizontal"
      android:contentDescription="@string/logo_small"
      android:scaleType="fitCenter"/>
  </LinearLayout>

</ScrollView>
```

(from Bitmaps/InSampleSize/app/src/main/res/layout/sample.xml)

- Uses a private `load()` method to load the images at the desired `inSampleSize` using a `BitmapFactory.Options` object
- Pours the images each into two `ImageView` widgets, one large and one small
- Updates some `TextView` widgets in the fragment to show how much memory those images are consuming

```
package com.commonsware.android.bitmap.iss;

import android.annotation.TargetApi;
import android.app.Fragment;
import android.content.Context;
```

```
import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Build;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.TextView;
import java.io.IOException;

public class BitmapFragment extends Fragment {
  private static final String KEY_SAMPLE_SIZE="inSampleSize";
  private AssetManager assets=null;

  static BitmapFragment newInstance(int inSampleSize) {
    BitmapFragment frag=new BitmapFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_SAMPLE_SIZE, inSampleSize);
    frag.setArguments(args);

    return(frag);
  }

  static String getTitle(Context ctxt, int inSampleSize) {
    return(String.format(ctxt.getString(R.string.title), inSampleSize));
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.sample, container, false);
    int inSampleSize=getArguments().getInt(KEY_SAMPLE_SIZE, 1);

    try {
      Bitmap flower=
          load("Tibouchina_urvilleana_flower_ja.jpg", inSampleSize);
      Bitmap logo=load("square.png", inSampleSize);

      ImageView iv=(ImageView)result.findViewById(R.id.flower_large);

      iv.setImageBitmap(flower);
      iv=(ImageView)result.findViewById(R.id.flower_small);
      iv.setImageBitmap(flower);
```

```
      iv=(ImageView)result.findViewById(R.id.logo_large);
      iv.setImageBitmap(logo);
      iv=(ImageView)result.findViewById(R.id.logo_small);
      iv.setImageBitmap(logo);

      TextView tv=(TextView)result.findViewById(R.id.byte_count);

      tv.setText(String.valueOf(byteCount(flower)));
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception loading bitmap", e);
    }

    return(result);
  }

  private Bitmap load(String path, int inSampleSize) throws IOException {
    BitmapFactory.Options opts=new BitmapFactory.Options();

    opts.inSampleSize=inSampleSize;

    return(BitmapFactory.decodeStream(assets().open(path), null, opts));
  }

  private AssetManager assets() {
    if (assets == null) {
      assets=getActivity().getAssets();
    }

    return(assets);
  }

  @TargetApi(Build.VERSION_CODES.KITKAT)
  private int byteCount(Bitmap b) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
      return(b.getAllocationByteCount());
    }

    return(b.getByteCount());
  }
}
```

(from Bitmaps/InSampleSize/app/src/main/java/com/commonsware/android/bitmap/iss/BitmapFragment.java)

If you run this on a device, you will see the images at the various sample sizes, one sample size per page of the `ViewPager`. While the quality of the loaded images

decreases as `inSampleSize` increases, the smaller `ImageView` widgets are still usable for the flower JPEG, though the line-art PNG suffers.

NOTE: The following screenshots will themselves be modified as part of the publishing process of the book and are here only for illustration purposes. You will want to run the demo and see the results first-hand.

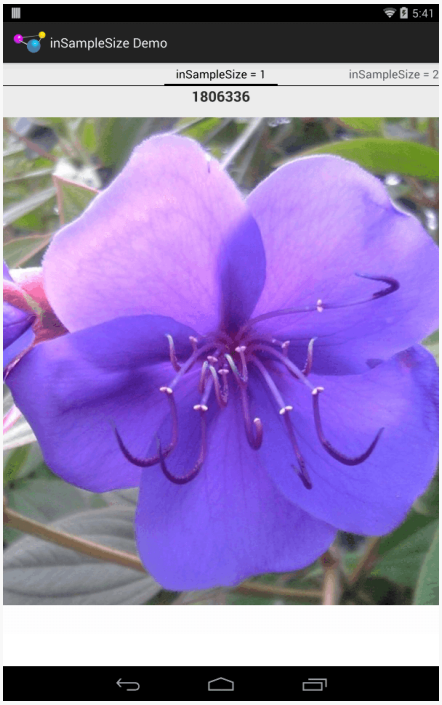


Figure 1021: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 1, Flower JPEG, Full Size

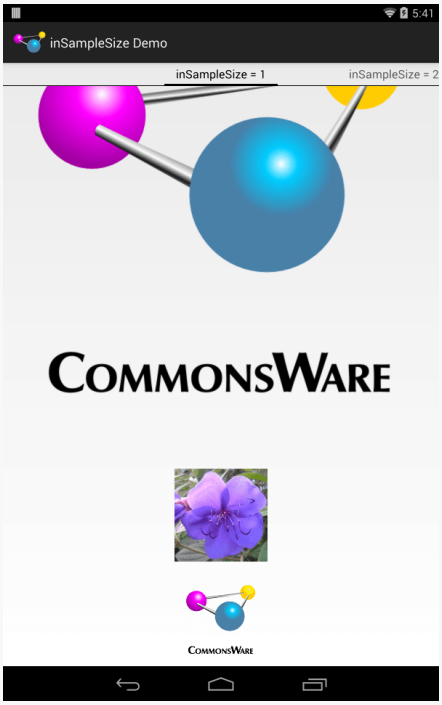


Figure 1022: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 1, CW Logo PNG (Full Size) and Smaller Sizes

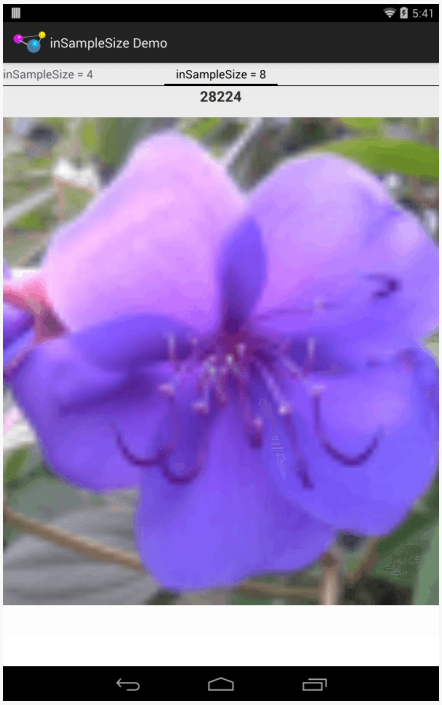


Figure 1023: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 8, Flower JPEG, Full Size

Figure 1024: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 8, CW Logo PNG (Full Size) and Smaller Sizes

The key, though, is the reduced memory footprint. The images loaded without sampling (`inSampleSize` of 1) take up 1,806,336 bytes of heap space (672 x 672 x 4 bytes per pixel). The `inSampleSize` of 8, by contrast, take up 28,244 bytes of heap space, less than 2% of the original.

You should consider experimenting with `inSampleSize` and determine an appropriate sampling level for the types of images you will receive (photos work better than line art) and the sizes you intend to use them in.

Bitmap Color Space

`BitmapFactory` will load images as `ARGB_8888` by default. That means that each pixel takes up four bytes, one each for the red, green, and blue color channels, plus a byte for the alpha channel (transparency).

However, particularly for thumbnails of photographs, where transparency probably does not exist and the image is small when viewed by the user, four bytes per pixel may be overkill.

Instead, you can set `inPreferredConfig` of the `BitmapFactory.Options` to `RGB_565`, which uses only two bytes (five bits for red, six bits for green, five bits for blue, and no transparency). This will cut your memory consumption for the bitmap in half, with no loss of resolution (as you get with `inSampleSize`).

Bitmap Reuse

If you will be doing a lot of work with bitmaps, particularly bitmaps of the same size, an object pool can be of tremendous help to minimize heap fragmentation. You can reuse the same `Bitmap` over and over again, by supplying it via `inBitmap` in the `BitmapFactory.Options` object. If the `Bitmap` is compatible with what you are looking to decode, it will be reused, rather than have a new `Bitmap` (backed by a new hunk of heap space) be created.

Here, "compatible" means:

- The image is the same bit depth configuration (`ARGB_8888` versus `RGB_565`)
- For API Level 18 and below, the resolution is identical; for API Level 19+, the `inBitmap` resolution is the same as or higher than the bitmap to be loaded

Releasing SQLite Memory

SQLite maintains a "page cache" of loaded pages from your database files. Curiously, it does so on a static basis, not on a per-`SQLiteDatabase` basis. Hence, even after you have closed your databases, you might still be consuming more memory than you need to, due to this cache.

From `onTrimMemory()`, you can call the static `releaseMemory()` method on SQLite, to try to free up some of this memory. This should not cause any database errors, but it may slow down the next few database accesses, as the necessary pages may no longer be cached and may have to be loaded again from disk.

Cheating

All your efforts at improving memory management may be merely "rearranging deck chairs on the Titanic". Certain scenarios simply require a lot of system RAM, such as complex image manipulations.

`android:largeHeap="true"` is one example of "cheating": working around the heap limits. However, as noted above, you may or may not get a particularly "large" heap, depending upon device capabilities.

The NDK is another option for cheating. The heap limits are for the Dalvik and ART runtime engines. Anything you do in native C/C++ code does not count against that heap limit. Hence, you might consider migrating complex logic into NDK code not only to get a possible boost in execution speed but also to avoid impacting your heap limit.

However, even with the NDK, you may not have enough RAM, because the *system* may not have enough RAM. Android One and similar low-spec devices might have as little as 512MB for the entire *device*, and your app would only be able to use a fraction of that, even from native code. Note that Android devices do not use "swap space" or similar memory paging techniques, and so once system RAM is exhausted, the device is likely to crash.

You can use `isLowRamDevice()` on `ActivityManager` to determine whether the device that your app is running on is considered to have low RAM. Nowadays, that means 512MB or lower of system RAM with a low screen resolution (e.g., 800x480). Before trying to use the NDK to cheat, check whether the device is a low-RAM device. If it is, you may need to disable certain features, rather than potentially crash the system by consuming all available system RAM.

The 1MB IPC Transaction Limit

Sometimes, even cheating cannot help you.

One example of this is a `FAILED BINDER TRANSACTION` error message, sans stack trace, that may show up in Logcat associated with a subsequent crash of yours. Alternatively, it sometimes appears as a `TransactionTooLargeException`. This message arises because there is a 1MB limit on the amount of memory used in IPC transactions. This includes:

- Starting activities or services, where the limit is imposed on the `Intent` and its extras
- Sending a broadcast, again measured in terms of the `Intent` and its extras
- the saved instance state `Bundle`
- responses from bound services (return values, `out` parameters)
- responses from `startActivityForResult()` calls

- and so on

Worse, this is not a per-transaction limit, but rather a per-process limit. *Usually*, you will only have one IPC request going at a time, but that is not always the case (e.g., you try starting an activity at the same time that you happen to be receiving a broadcast). Hence, coming anywhere close to the 1MB limit is risky, lest your transaction combine with others to have you exceed the 1MB limit.

This limit is baked into the operating system and cannot be altered by developers.

Worse, you do not actually know if the transaction totally failed (it could not be sent) or partially failed (the transaction was sent, but the response failed).

Overall, try to keep your transaction sizes low:

- Do not put bitmaps into transactions wherever possible
- Do not transfer large arrays, huge strings, and the like
- Do not use IPC when in-process communications (e.g., event buses) can work

Finding Memory Leaks

Android Studio's heap analyzer is your #1 tool for identifying memory leaks and the culprits behind running out of heap space. Particularly when used with Android 3.0+ versions of Android, the heap analyzer can tell you:

1. Who are the major sources of memory consumption, both directly (e.g., bitmaps) or indirectly (e.g., leaked activities holding onto lots of widgets)
2. What is keeping objects in memory unexpectedly, defying standard garbage collection — the way that you leak memory in a managed runtime environment like Dalvik or ART

Android Studio's heap analyzer builds on the earlier Memory Analysis Tool (MAT), used by Java developers, and by Android developers prior to Android Studio.

However, Android Studio's heap analysis leaves a lot to be desired. Not only do you have to manually examine and check heap dumps, but you get a lot of false positives due to bugs in Android. A library that helps with both of these issues is LeakCanary, and we will examine it in this chapter as well.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Android's process model. Reading the introductory chapter to this trail might be nice.

Android Studio Profiler

The first question is: when do we bother looking for leaks? Complex apps are complex, and so we might spend a lot of time looking for leaks that either do not exist or do not matter much.

In Android Studio, the Android Profiler tool will allow you to examine the real-time behavior of your app with respect to various system resources, such as heap space in your app:

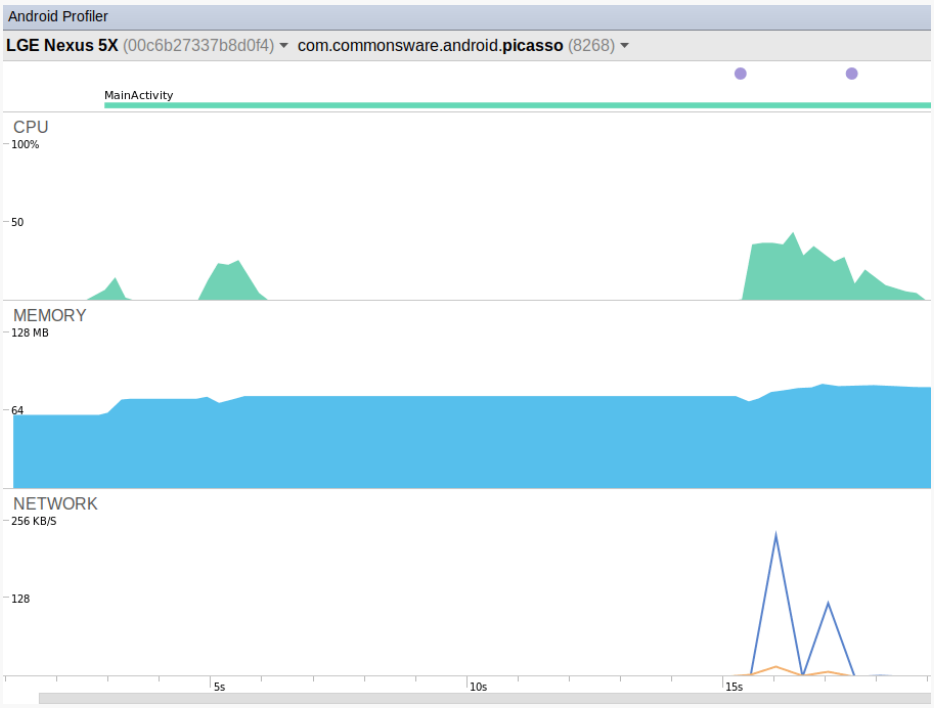


Figure 1025: Android Profiler

Clicking on the memory portion of the graphs will bring up details for the memory consumption:

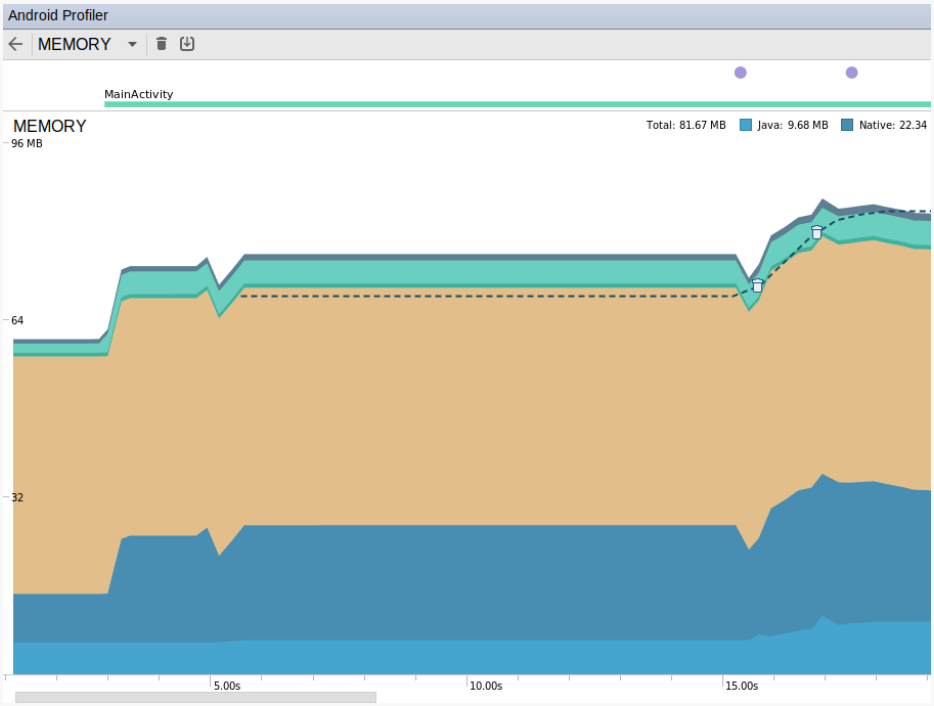


Figure 1026: Android Profiler, Showing Memory Details

The color coding shows the different types of memory being consumed. For example, the largest area in this graph is the tan "Graphics" area. The sample app being tested here happens to be a Picasso sample app from the chapter on Internet access. However, "Graphics" does not refer to bitmaps, but rather is "used for graphics buffer queues to display pixels to the screen, including GL surfaces, GL textures, and so on", according to the documentation. Generally speaking, you will be most interested in:

- The light blue portion of the graph, representing memory consumed by your Java objects
- The dashed line, which represents the number of allocated Java objects
- The garbage can icons, representing garbage collection events

On Android 5.0+ devices, the memory usage can also fall... while your app is no longer in the foreground:

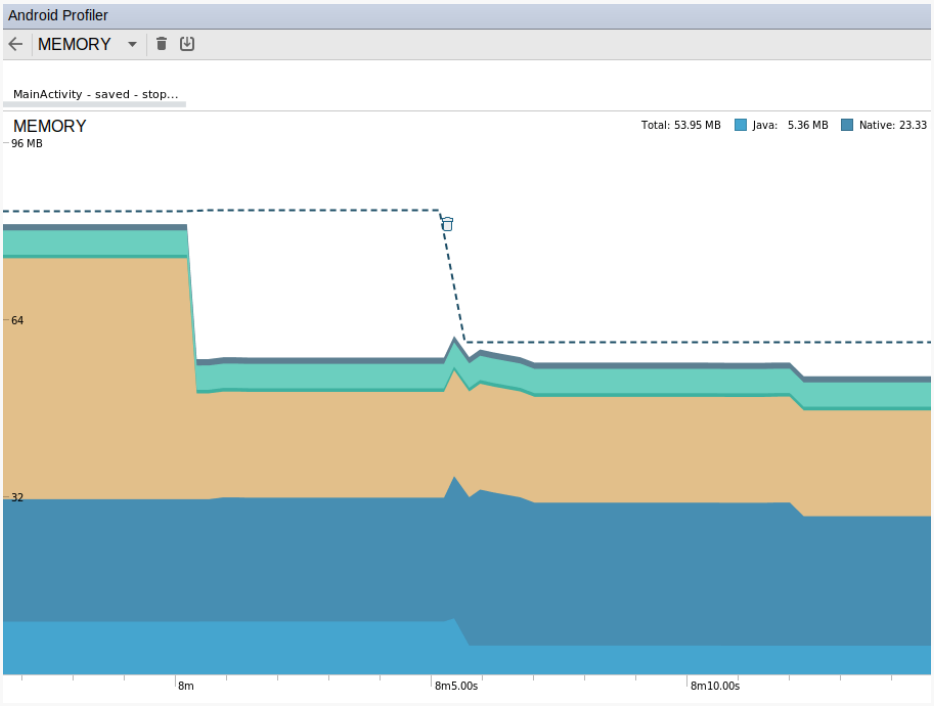


Figure 1027: Android Studio, Android Monitor, Memory Tab, Showing Shrunken Heap

The major drop in the memory usage came with the release of some of those GL buffers and textures, some of which are no longer necessary when the app no longer is in the foreground from a UI standpoint.

The major drop in the number of allocated Java objects presumably came from ART. Once your app is no longer in the foreground, ART will do a more aggressive garbage collection run, including moving objects in heap space to coalesce free blocks. If this frees up some of the allocated pages from the OS, ART can then free those pages, returning the memory to the OS and reducing our app's overall memory footprint.

You can also perform a manual garbage collection run by tapping the "garbage can" icon in the toolbar. Other toolbar buttons include:

- A button to dump the heap (discussed later in this chapter)
- A close button to close this tool
- Buttons to zoom in, zoom out, or reset the zoom to its default level
- A "Live" button that serves to pause the results or, from a paused state, resume normal real-time output

Getting Heap Dumps

The first step to analyzing what is in your heap is to actually get your hands on what is in your heap. This is referred to as creating a "heap dump" — what amounts to a log file containing all your objects and who points to what.

In Android Studio

In the memory details view of the Android Profiler tool in Android Studio, you can create a heap dump by clicking the toolbar button that looks like an open square with a downwards-pointing arrow in it, adjacent to the "garbage can" icon for forcing garbage collection:

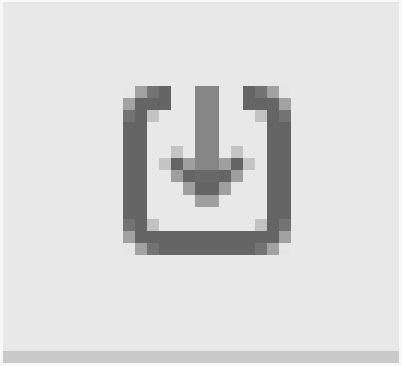


Figure 1028: "Dump Java Heap" Toolbar Button in Android Profiler

Tap that, then go get a cup of coffee (or another preferred beverage). Generating a heap dump used to take a few seconds. With Android Studio 3.0, it takes much longer than that.

Eventually the bottom portion of the Android Profiler pane will show the results of the heap dump.

From Code

Another possibility is to trigger the heap dump yourself from code. The `dumpHprofData()` static method on the `Debug` class (in the `android.os` package) will write out a heap dump to the file you indicate. Since you will need to transfer them off the device or emulator, it will be simplest to specify a path to a file on external storage, which means that your project will need the `WRITE_EXTERNAL_STORAGE` permission.

To view the results in Android Studio, you will need to transfer the file from wherever you saved it on the device or emulator to your development machine. For example, you can double-click on it in the Device File Explorer.

The UI that you get has the same basic functionality as does the UI from a manually-requested heap dump from the Android Profiler, though they look somewhat different.

Analyzing Heap Dumps in Android Studio

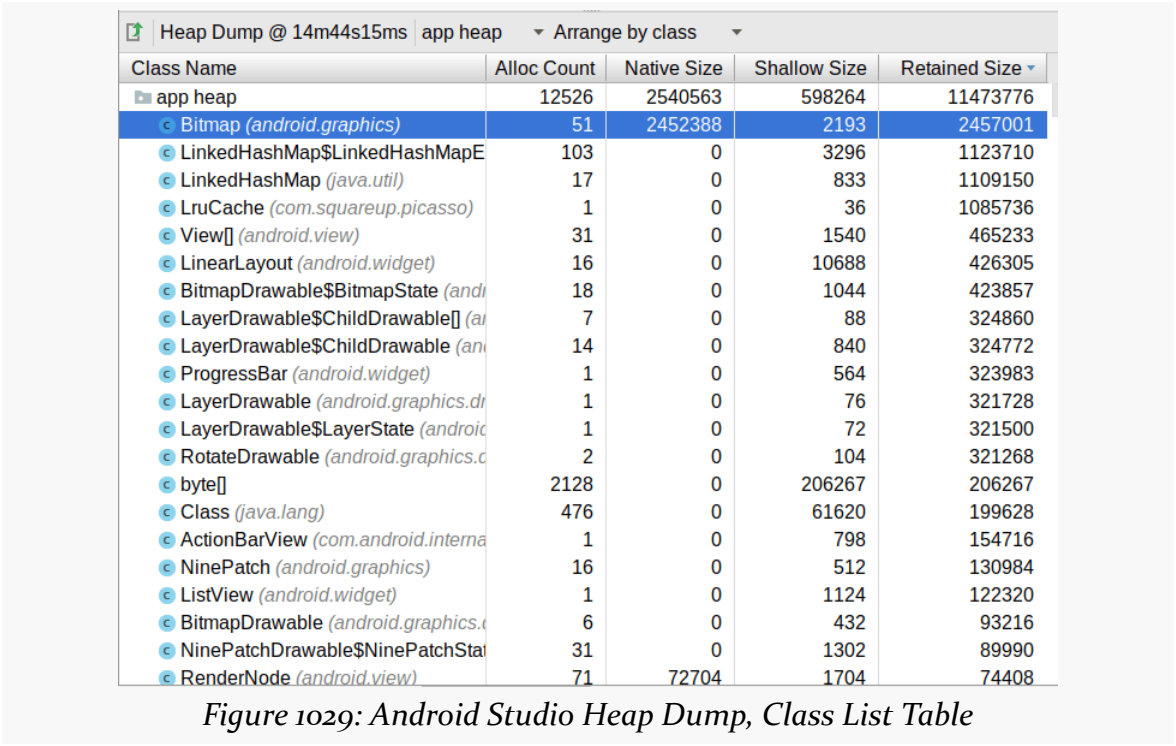
Having a heap dump is nice, but we need tools to determine exactly what is in there and what that means for our app. Android Studio lets us examine a heap dump to see what is going on and perhaps identify leaks.

Navigating the Heap Dump UI

There are several pieces to the UI that we use to examine a heap dump.

Class List

We start off with a class list table:



Class Name	Alloc Count	Native Size	Shallow Size	Retained Size
app heap	12526	2540563	598264	11473776
Bitmap *(android.graphics)*	51	2452388	2193	2457001
LinkedHashMap$LinkedHashMapE	103	0	3296	1123710
LinkedHashMap *(java.util)*	17	0	833	1109150
LruCache *(com.squareup.picasso)*	1	0	36	1085736
View[] *(android.view)*	31	0	1540	465233
LinearLayout *(android.widget)*	16	0	10688	426305
BitmapDrawable$BitmapState *(andr*	18	0	1044	423857
LayerDrawable$ChildDrawable[] *(a*	7	0	88	324860
LayerDrawable$ChildDrawable *(an*	14	0	840	324772
ProgressBar *(android.widget)*	1	0	564	323983
LayerDrawable *(android.graphics.dr*	1	0	76	321728
LayerDrawable$LayerState *(androic*	1	0	72	321500
RotateDrawable *(android.graphics.c*	2	0	104	321268
byte[]	2128	0	206267	206267
Class *(java.lang)*	476	0	61620	199628
ActionBarView *(com.android.interna*	1	0	798	154716
NinePatch *(android.graphics)*	16	0	512	130984
ListView *(android.widget)*	1	0	1124	122320
BitmapDrawable *(android.graphics.c*	6	0	432	93216
NinePatchDrawable$NinePatchStat	31	0	1302	89990
RenderNode *(android.view)*	71	72704	1704	74408

Figure 1029: Android Studio Heap Dump, Class List Table

This table comes filled in with a list of classes and primitive arrays, sorted by "retained size". This indicates how much memory those objects, and everything that they point to, consume. So, for example, this heap dump contains 51 `Bitmap` objects with a retained size of 2,457,001 bytes.

The other columns are:

- Alloc Count: the number of instances of this class found in your app's heap
- Native Size: the amount of "native memory" used by these objects (generally zero, but non-zero for some classes like `Bitmap`)
- Shallow Size: how much memory these instances consume in their own primitives, not including any other objects that they point to

Roughly speaking, the sum of the "Native Size" and the "Shallow Size" equals the amount of space that the objects are taking up directly themselves, not counting references to other objects.

Heap Selector

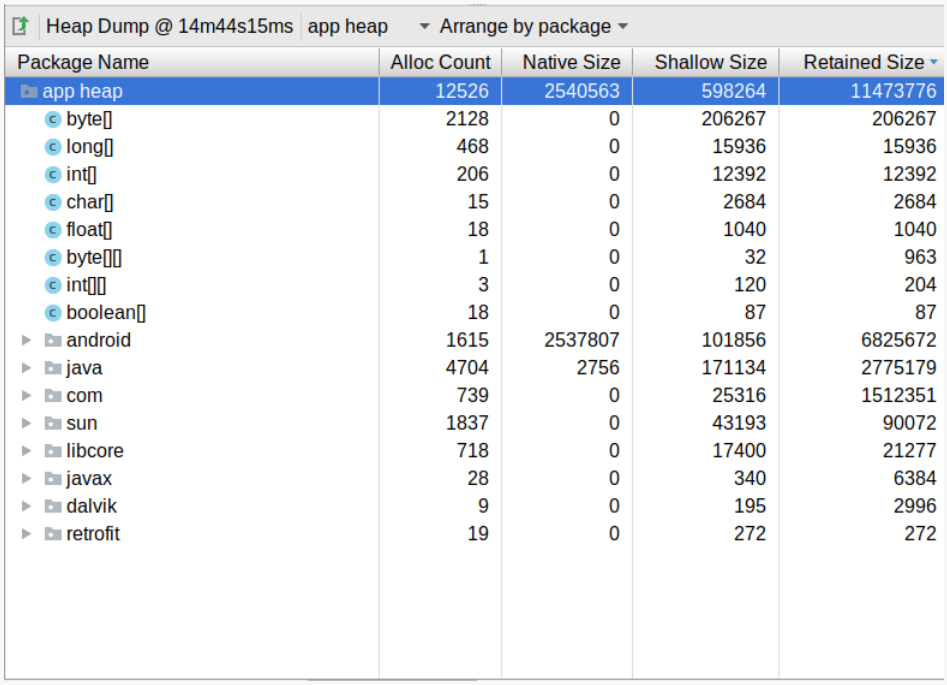
The drop-down above the table that defaults to "App heap" will have other options on Android 5.0+ devices. Specifically, you can switch between the app heap, the undocumented "default heap", the similarly-undocumented "image heap", and the equally-undocumented "zygote heap". The zygote is a core OS process, started when the device boots; all Android SDK apps are forked off of the zygote. Given that and other announced ART tidbits suggests that:

- the "image heap" may be the large object space, mostly set aside for bitmaps
- the "zygote heap" may be the objects in the heap that were instantiated by the zygote and its initialization of the Android framework classes, as opposed to your code

In general, you want the app heap, which is what appears by default.

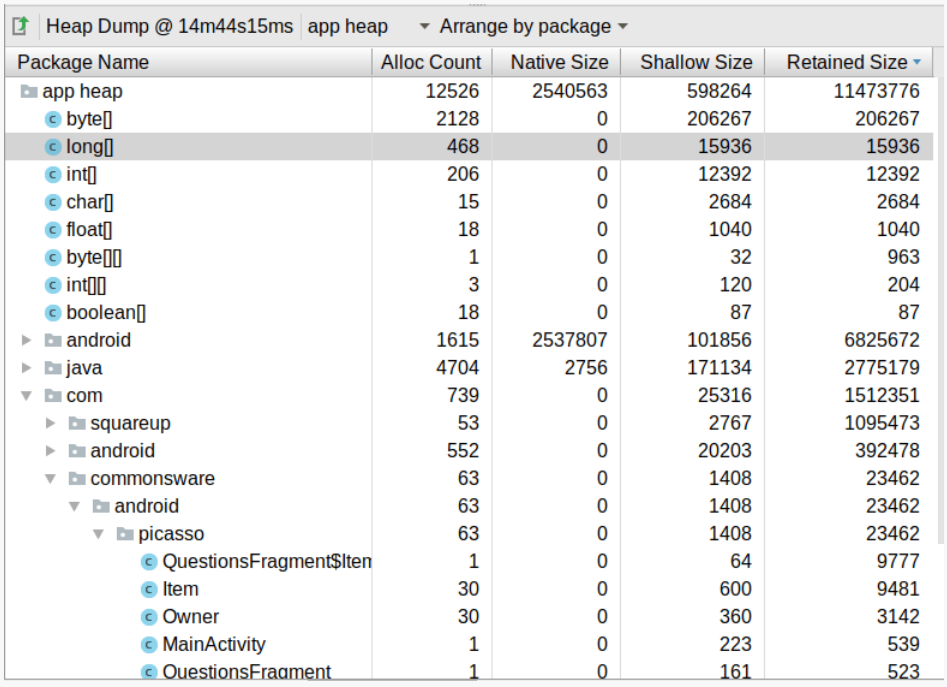
Package Tree View

The drop-down above the table that defaults to "Arrange by class" can be toggled to "Arrange by package", which turns the table into a tree-table, for navigation by Java package name, plus primitive arrays:



Package Name	Alloc Count	Native Size	Shallow Size	Retained Size
app heap	12526	2540563	598264	11473776
byte[]	2128	0	206267	206267
long[]	468	0	15936	15936
int[]	206	0	12392	12392
char[]	15	0	2684	2684
float[]	18	0	1040	1040
byte[][]	1	0	32	963
int[][]	3	0	120	204
boolean[]	18	0	87	87
android	1615	2537807	101856	6825672
java	4704	2756	171134	2775179
com	739	0	25316	1512351
sun	1837	0	43193	90072
libcore	718	0	17400	21277
javax	28	0	340	6384
dalvik	9	0	195	2996
retrofit	19	0	272	272

Figure 1030: Android Studio Heap Dump, Package Tree View, As Initially Launched



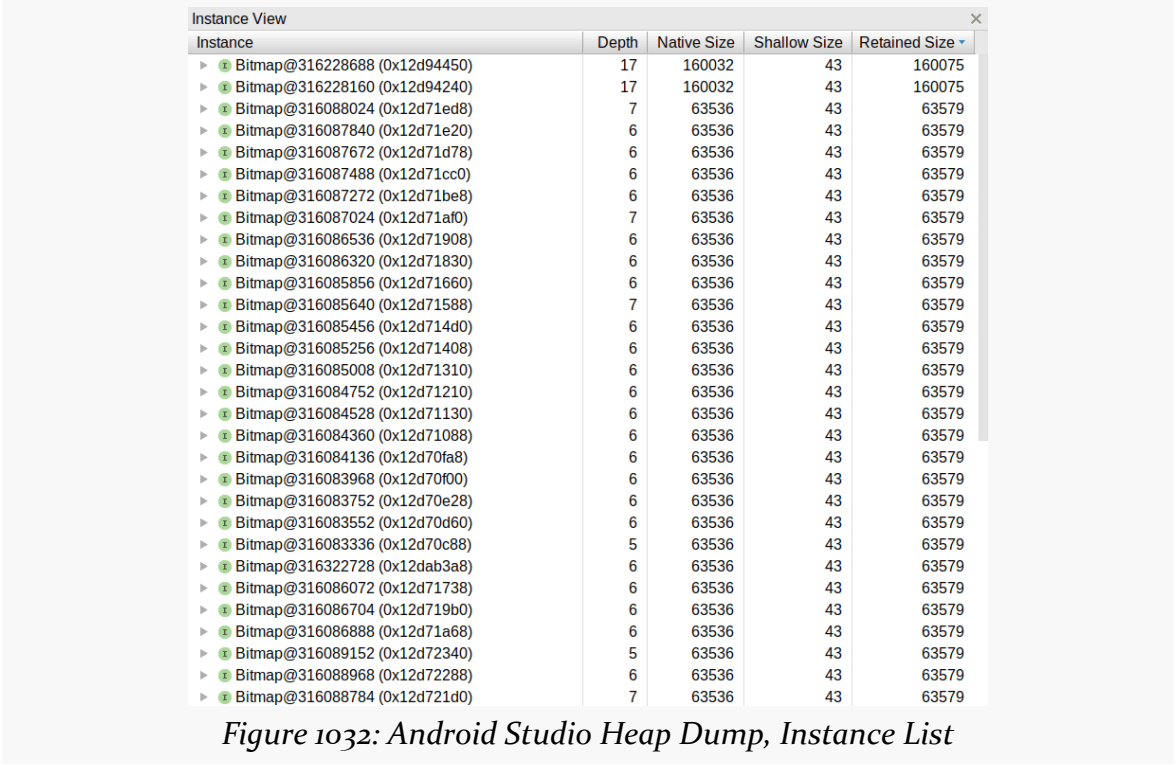
Package Name	Alloc Count	Native Size	Shallow Size	Retained Size
app heap	12526	2540563	598264	11473776
byte[]	2128	0	206267	206267
long[]	468	0	15936	15936
int[]	206	0	12392	12392
char[]	15	0	2684	2684
float[]	18	0	1040	1040
byte[][]	1	0	32	963
int[][]	3	0	120	204
boolean[]	18	0	87	87
android	1615	2537807	101856	6825672
java	4704	2756	171134	2775179
com	739	0	25316	1512351
squareup	53	0	2767	1095473
android	552	0	20203	392478
commonsware	63	0	1408	23462
android	63	0	1408	23462
picasso	63	0	1408	23462
QuestionsFragment$Iten	1	0	64	9777
Item	30	0	600	9481
Owner	30	0	360	3142
MainActivity	1	0	223	539
QuestionsFragment	1	0	161	523

Figure 1031: Android Studio Heap Dump, Package Tree View, Drilled Down Into Packages

This view will make it easier for you to find classes from your app or from well-known libraries, since those classes will be clustered into their own pacakges.

Instance List

If you click on a class in either the class list view or the package tree view, a table on the right will show a list of the instances of that class that were found in your heap:


Instance View

Instance	Depth	Native Size	Shallow Size	Retained Size
Bitmap@316228688 (0x12d94450)	17	160032	43	160075
Bitmap@316228160 (0x12d94240)	17	160032	43	160075
Bitmap@316088024 (0x12d71ed8)	7	63536	43	63579
Bitmap@316087840 (0x12d71e20)	6	63536	43	63579
Bitmap@316087672 (0x12d71d78)	6	63536	43	63579
Bitmap@316087488 (0x12d71cc0)	6	63536	43	63579
Bitmap@316087272 (0x12d71be8)	6	63536	43	63579
Bitmap@316087024 (0x12d71af0)	7	63536	43	63579
Bitmap@316086536 (0x12d71908)	6	63536	43	63579
Bitmap@316086320 (0x12d71830)	6	63536	43	63579
Bitmap@316085856 (0x12d71660)	6	63536	43	63579
Bitmap@316085640 (0x12d71588)	7	63536	43	63579
Bitmap@316085456 (0x12d714d0)	6	63536	43	63579
Bitmap@316085256 (0x12d71408)	6	63536	43	63579
Bitmap@316085008 (0x12d71310)	6	63536	43	63579
Bitmap@316084752 (0x12d71210)	6	63536	43	63579
Bitmap@316084528 (0x12d71130)	6	63536	43	63579
Bitmap@316084360 (0x12d71088)	6	63536	43	63579
Bitmap@316084136 (0x12d70fa8)	6	63536	43	63579
Bitmap@316083968 (0x12d70f00)	6	63536	43	63579
Bitmap@316083752 (0x12d70e28)	6	63536	43	63579
Bitmap@316083552 (0x12d70d60)	6	63536	43	63579
Bitmap@316083336 (0x12d70c88)	5	63536	43	63579
Bitmap@316322728 (0x12dab3a8)	6	63536	43	63579
Bitmap@316086072 (0x12d71738)	6	63536	43	63579
Bitmap@316086704 (0x12d719b0)	6	63536	43	63579
Bitmap@316086888 (0x12d71a68)	6	63536	43	63579
Bitmap@316089152 (0x12d72340)	5	63536	43	63579
Bitmap@316088968 (0x12d72288)	6	63536	43	63579
Bitmap@316088784 (0x12d721d0)	7	63536	43	63579

Figure 1032: Android Studio Heap Dump, Instance List

The “shallow size” refers to the number of bytes consumed directly by that particular instance, such as by primitive fields. The “dominating size” roughly equates to “how much memory can this object be blamed for”. In other words, if that object could be garbage-collected, how much would we recover, not only from the “shallow size” but from other objects uniquely referenced by this object?

The “depth” refers to how many hops away from a garbage collection root (“GC root”) this object is.

This table initially appears as a simple table. In reality, though, it is a tree table. You can expand nodes in the tree to drill down into all the objects referenced by a particular instance, and all objects that reference the instance in question:

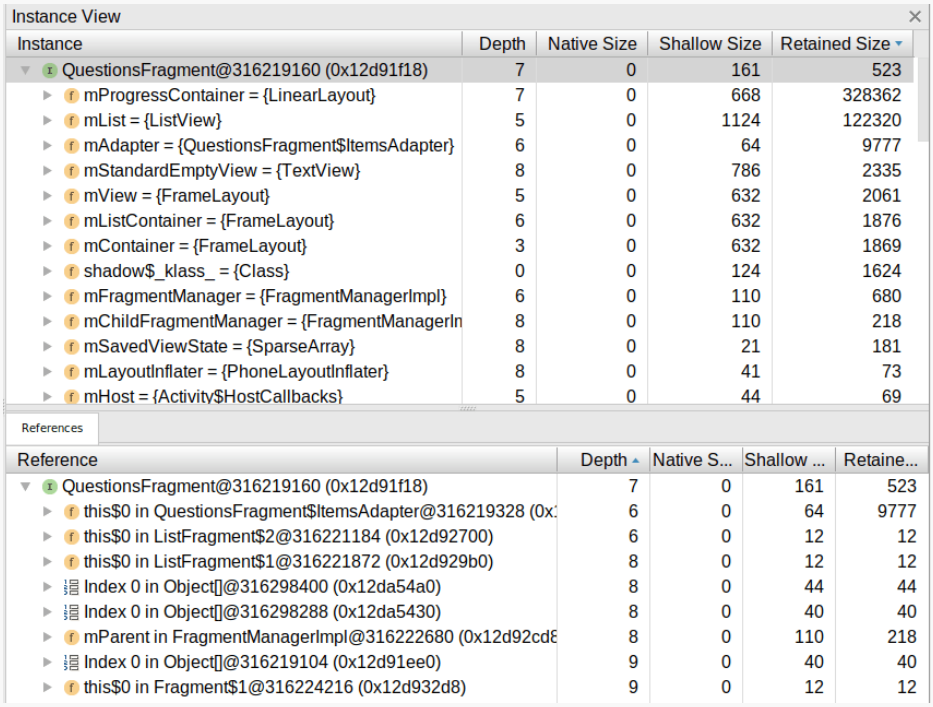


Figure 1033: Android Studio Heap Dump, Instance Tree

You can further expand the References tree to see who references some of those references, and so on.

Identifying Leak Candidates

All of that is just great, but you still need to determine if you have a memory leak and, if so, where is it coming from.

Basically, you rummage through the class list or package tree, looking for classes that either:

- you would not expect to be there (e.g., all instances should have been garbage-collected)
- you would not expect to be so numerous
- you would not expect to retain so much heap space

Bear in mind that the act of generating a heap dump only logs objects that are reachable from other objects, or themselves are considered "garbage collection roots" (a.k.a., "GC roots"). Any objects that are actual garbage, but perhaps have not

yet been collected by the garbage collector, do not appear in the dump. Hence, if you see it in the heap snapshot tab, the objects are "real", not uncollected garbage.

Conversely, just because you find an object in the heap does not mean that it is truly "leaked". For example:

- Activities that have not been destroyed are not leaked, strictly speaking, though you may wish to consider whether changes to your app's navigation can allow you to reuse existing activity instances better.
- Objects that are part of a cache, such as Picasso's memory caching of downloaded images, are intentionally "leaked". You may use what you see in the heap snapshot tab to elect to reduce the size of those caches, or perhaps better consolidate multiple disparate caches, where possible.
- Objects in use by a running thread are not leaked... unless the thread itself is effectively leaked (i.e., exists, and refers to objects, but you do not know why that thread is still outstanding).

Common Leak Scenarios

With all that in mind, let's look at a few common scenarios of leaking objects, to see what those leaks look like when we do a heap dump and analyze that dump in Android Studio.

The Static Widget

The Leaks/StaticWidget sample project does something naughty:

```
package com.commonsware.android.button;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;

public class ButtonDemoActivity extends Activity {
  private static Button pleaseDoNotDoThis;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    pleaseDoNotDoThis=(Button)findViewById(R.id.button1);
```

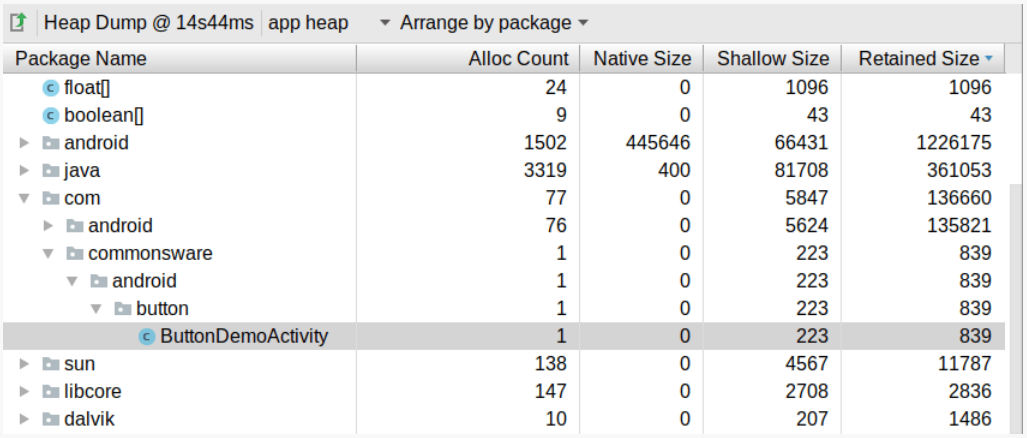
```
  }
}
```

(from Leaks/StaticWidget/app/src/main/java/com/commonsware/android/button/ButtonDemoActivity.java)

We take a widget (specifically a `Button`) and put it in a static data member, and never replace it with null.

As a result, even if the user presses BACK to get out of the activity, the static data member holds onto `Button`, which itself has a reference back to our `Activity`.

If you run the app, press BACK to exit the activity, and generate a heap dump of this process, you will see that `ButtonDemoActivity` appears in the dump:



Package Name	Alloc Count	Native Size	Shallow Size	Retained Size
float[]	24	0	1096	1096
boolean[]	9	0	43	43
android	1502	445646	66431	1226175
java	3319	400	81708	361053
com	77	0	5847	136660
android	76	0	5624	135821
commonsware	1	0	223	839
android	1	0	223	839
button	1	0	223	839
ButtonDemoActivity	1	0	223	839
sun	138	0	4567	11787
libcore	147	0	2708	2836
dalvik	10	0	207	1486

Figure 1034: Heap Dump UI, Showing Leaked Activity

If you click on `ButtonDemoActivity` and poke around the References tab, you will find where our `static` field shows up:

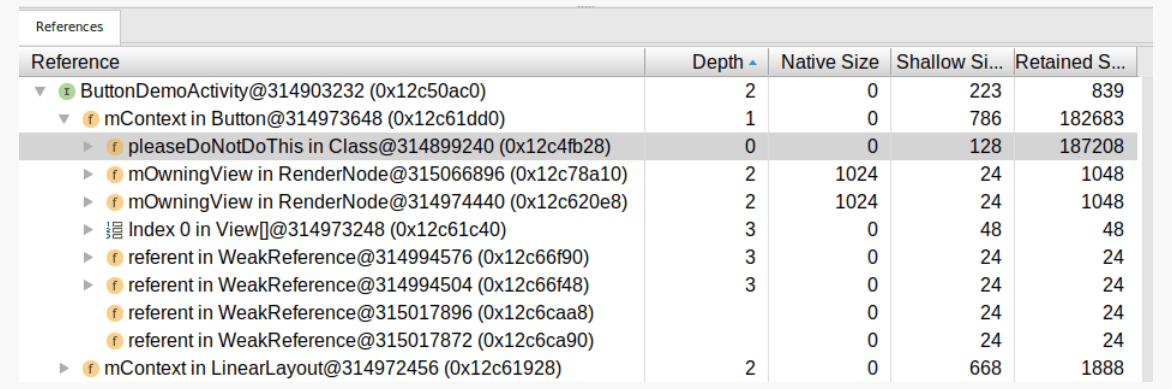


Figure 1035: Heap Dump UI, Showing Static Field

The leaked object (`ButtonDemoActivity`) is referenced by an `mContext` field in a static `pleaseDoNotDoThis` field in `ButtonDemoActivity` itself. The latter item has a depth of 0, so we know that it is a GC root. The hope is that you will recognize some of the items shown here (e.g., field names like `pleaseDoNotDoThis`) and can see how those items affect the ability for Android to garbage collect the leaked object.

Thread References

The Leaks/LeakedThread sample project does something else naughty:

```
package com.commonsware.android.leak.thread;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;

public class LeakedThreadActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    new Thread() {
      public void run() {
        while(true) {
          SystemClock.sleep(100);
        }
      }
    }.start();
  }
}
```

(from Leaks/LeakedThread/app/src/main/java/com/commonsware/android/leak/thread/LeakedThreadActivity.java)

Here, we kick off a `Thread` from `onCreate()` of our activity and have it enter a pseudo-polling loop, sleeping for 100ms per pass through the loop.

This is naughty for all sorts of reasons:

- Fast polling loops like this are bad for the battery
- We start a thread and never stop it
- We are using an anonymous inner class for our `Thread`

The latter two flaws combine to cause a memory leak.

So, we can go through the package tree view, find the Java packages for the code, and see what objects from those packages are outstanding:

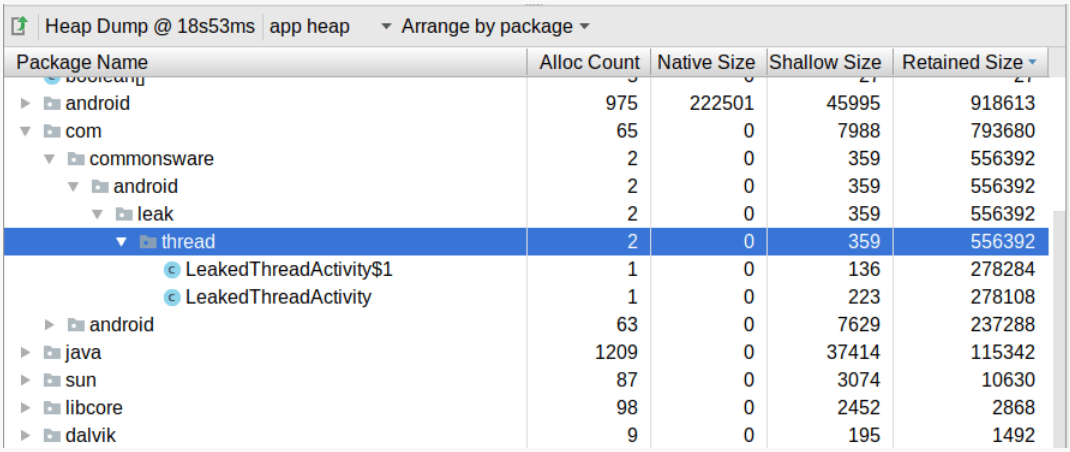


Figure 1036: Heap Dump UI, Showing Classes In App Package

Here, we see that we have leaked two objects. One is `LeakedThreadActivity`. The other is an anonymous inner class of `LeakedThreadActivity` (assigned the name `LeakedThreadActivity$1` by the Java compiler).

Clicking on the activity and examining the first child in the reference tree once again discloses the leak:

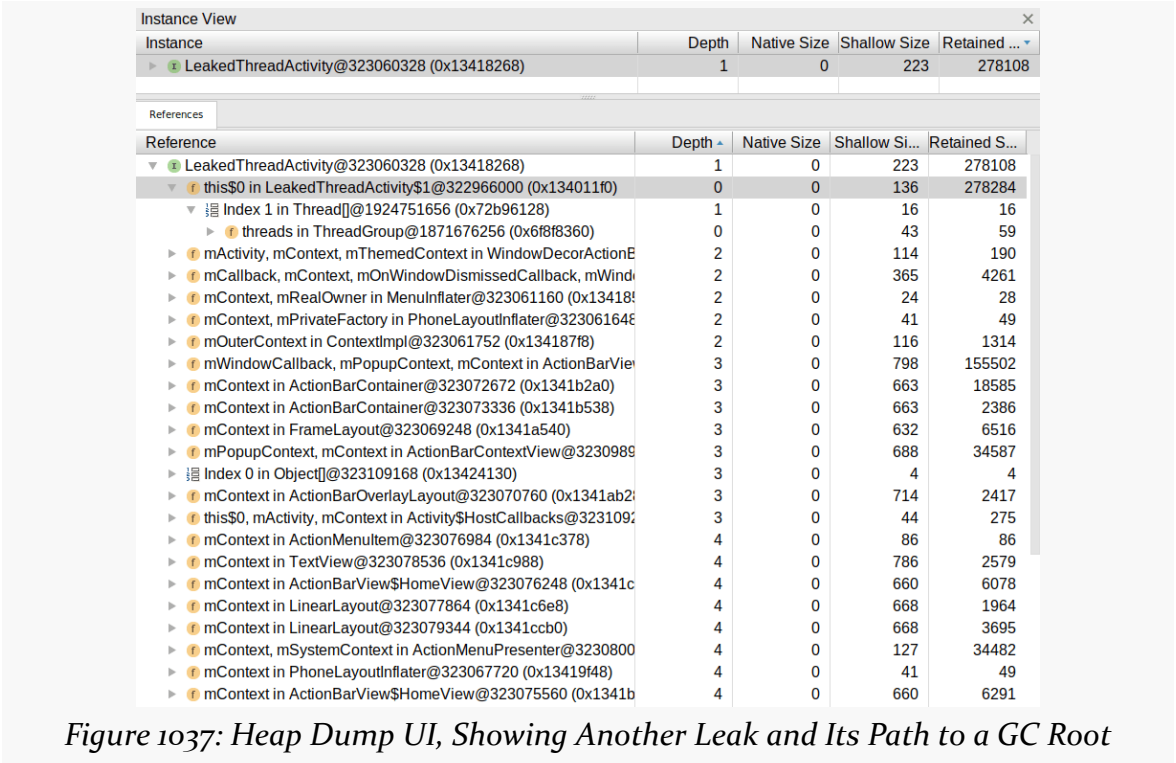


Figure 1037: Heap Dump UI, Showing Another Leak and Its Path to a GC Root

Our zero-depth entry is `threads`, which is basically the collection of all Java `Thread` objects that are still alive in this process. One of those is our anonymous inner class (`this$0` in `LeakedThreadActivity$1`), which holds onto the activity instance.

To avoid this sort of leak:

- Do not have everlasting threads — whatever component creates a thread needs to stop the thread when the component is being destroyed
- Do not use anonymous inner classes when creating threads, as an anonymous inner class has an implicit reference back to the outer class instance that created it (in this case, our activity), and that outer class instance cannot be garbage-collected until the thread terminates

Retaining Too Much

In the chapter on threads, we had an `AsyncTask` demo app that used a retain fragment to manage the task. That fragment was a `ListFragment`, and it was responsible for displaying the Latin words as those words were "downloaded" in the background by the task. Google is not a fan of retained fragments having widgets... and the `Leaks/ConfigChange` sample project demonstrates why.

The change in this project versus the original mostly comes down to a humble `Button`, which we will use to restart the download from the beginning once it has completed:

```
  <Button
    android:id="@+id/again"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/btn_again"/>
</LinearLayout>
```

(from Leaks/ConfigChange/app/src/main/res/layout/main.xml)

The `Button` itself is stored as a field in the fragment, named `btnAgain`. This already raises some concerns, if we are retaining the fragment. However, this approach is safe, if and only if we clear out or refresh that field on a configuration change. For example, if you used `findViewById()` to get the `Button` and assign it to `btnAgain` in `onViewCreated()`, you would not have a problem, as `onViewCreated()` is called as part of the configuration change, even for retained fragments.

However, this sample app instead lazy-initializes that data member, via a `getAgain()` getter method:

```
  private Button getAgain() {
    if (btnAgain==null) {
      btnAgain=(Button)getView().findViewById(R.id.again);
    }

    return(btnAgain);
  }
```

(from Leaks/ConfigChange/app/src/main/java/com/commonsware/android/leak/configchange/AsyncDemoFragment.java)

That getter method is used in the rest of the fragment to retrieve the `Button`, such as in `onViewCreated()`:

```
  @Override
  public void onViewCreated(View v, Bundle savedInstanceState) {
    super.onViewCreated(v, savedInstanceState);

    getListView().setScrollbarFadingEnabled(false);
    setListAdapter(adapter);

    getAgain().setOnClickListener(this);

    if (task!=null) {
```

```
    getAgain().setEnabled(false);
  }
}
```

(from Leaks/ConfigChange/app/src/main/java/com/commonsware/android/leak/configchange/AsyncDemoFragment.java)

...`onClick()` (as the fragment now implements the `View.OnClickListener` interface):

```
@Override
public void onClick(View v) {
  getAgain().setEnabled(false);
  adapter.clear();
  task=new AddStringTask();
  task.execute();
}
```

(from Leaks/ConfigChange/app/src/main/java/com/commonsware/android/leak/configchange/AsyncDemoFragment.java)

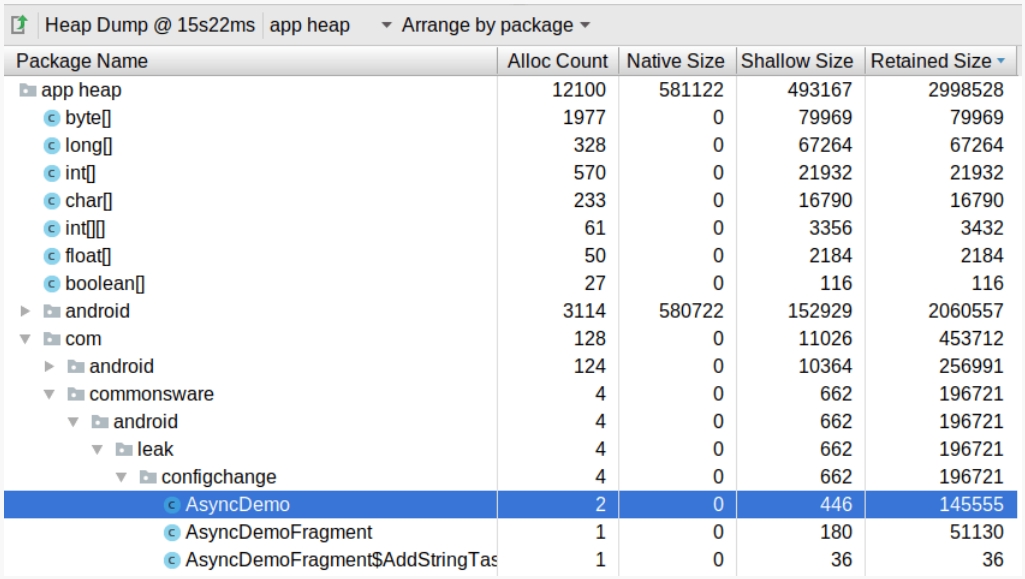
...and `onPostExecute()` of the `AsyncTask`:

```
  @Override
  protected void onPostExecute(Void unused) {
    task=null;
    getAgain().setEnabled(true);
  }
```

(from Leaks/ConfigChange/app/src/main/java/com/commonsware/android/leak/configchange/AsyncDemoFragment.java)

Nowhere do we set `btnAgain` to `null`, including after a configuration change. So, when the activity starts up, everything is fine. However, when we rotate the screen or otherwise undergo a configuration change, the fragment misbehaves. `getAgain()` says "hey, `btnAgain` is already initialized, so I can skip the `findViewById()` call". But we have a *different* `Button` now after the configuration change, and `btnAgain` is pointing to the original `Button`. That original `Button` is tied to the original, pre-configuration change `Activity` instance, and we have a leak, until the *second* `Activity` is destroyed.

If you run the app, rotate the screen, and then capture a heap dump, the snapshot will show two outstanding instances of `AsyncDemo`:



Package Name	Alloc Count	Native Size	Shallow Size	Retained Size
app heap	12100	581122	493167	2998528
byte[]	1977	0	79969	79969
long[]	328	0	67264	67264
int[]	570	0	21932	21932
char[]	233	0	16790	16790
int[][]	61	0	3356	3432
float[]	50	0	2184	2184
boolean[]	27	0	116	116
android	3114	580722	152929	2060557
com	128	0	11026	453712
android	124	0	10364	256991
commonsware	4	0	662	196721
android	4	0	662	196721
leak	4	0	662	196721
configchange	4	0	662	196721
AsyncDemo	2	0	446	145555
AsyncDemoFragment	1	0	180	51130
AsyncDemoFragment$AddStringTas	1	0	36	36

Figure 1038: Heap Snapshot Tab, Showing Two Activities Instead of One

However, this leak will be difficult to diagnose, for two reasons:

1. Android Studio's heap analyzer does a poor job of illustrating what is holding onto the activities
2. Not only are *you* leaking the activities, but so is Android itself, as will be explored in the next section

A Canary in a Leaky Coal Mine

When the author of this book was testing the previous section's demo, he was trying to use Android Studio to confirm that the leak was caused by the `Button`. As part of that analysis, he went back to the original `Threads/AsyncDemo` sample project... and Android Studio said that *it* was leaking the activity.

At this point, a long series of expletives could be heard emanating from the author's office.

To help try to suss out exactly what was going on, the author turned to a library that you may wish to consider: LeakCanary. And, as it turns out, LeakCanary indicates that the Android Studio-reported leak is a false positive, and that there is no serious memory leak.

Introducing LeakCanary

LeakCanary is another library from the indefatigable developers at Square. It allows you to monitor certain objects to see if they get leaked. In particular, if you use the standard setup, it will automatically watch for activities that get leaked. When it detects a leak, it will dump the heap, then read in the heap dump *on the device* and try to determine where the leak is coming from. To help with that, it has a roster of known false positives that it can filter out, and the authors encourage the community to provide more false positives where possible.

If a leak is detected, but it is a false positive, a message will be dumped to Logcat with the details. If a leak is detected that appears to be genuine, a `Notification` will appear, leading to an activity that will show you the source of the leak.

Adding LeakCanary to a Project

Adding LeakCanary to a project is fairly easy, courtesy of some well-designed defaults and a tricky use of build type-specific dependencies.

Deal with the Limitations

Unfortunately, LeakCanary has its own set of problems.

First, your project's `compileSdkVersion` needs to be 21 or higher. The author of LeakCanary elected to use `Theme.Material` on Android 5.0+ as the theme for the result activity. Even if you only plan on running your LeakCanary-enabled app on Android 4.4 or below, to compile successfully, you need to have `compileSdkVersion` set to 21 or higher, so that the reference to `Theme.Material` can be recognized. If this is a problem for your project (e.g., team decision to stick with an older `compileSdkVersion` for a while), you are welcome to try to fork LeakCanary and remove that theme.

LeakCanary also has bugs that you might trip over, including a bug that prevents it from working on low-memory environments, including some emulators.

Adding the Dependencies

We only want LeakCanary to be used in `debug` builds, not `release` builds. Even if we are leaking memory, the effects of LeakCanary (including slow heap dumps) are not the sort of thing that we should be putting users through.

Yet, at the same time, we will need a bit of Java code to hook up LeakCanary itself. Ordinarily, this would require setting up `src/debug/` and `src/release/` source sets and trying to isolate the LeakCanary-specific code to the `debug` build.

LeakCanary addresses this by publishing two versions of the artifact: the real one (for `debug`) and a `no-op` one (for `release`). The public API for each is identical, so your application code can build in either case. It just so happens that the `no-op` artifact does nothing in response to the API, as it merely contains stubs necessary to satisfy the API. This is *much* simpler, and for coarse-grained APIs is a technique worth emulating.

The `Leaks/AsyncTask` sample project is a clone of the `Threads/AsyncDemo` sample project, but uses LeakCanary. In its `app` module's `build.gradle` file, we have the twin dependencies, scoped for the appropriate build types:

```
dependencies {
    debugCompile 'com.squareup.leakcanary:leakcanary-android:1.5.1'
    releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.1'
}
```

(from Leaks/AsyncTask/app/build.gradle)

If you have your own custom build types, you would need to adjust the conditional dependencies to match, using the `no-op` one for any build that should not have the real LeakCanary in it.

Adding the Application

Usually, if you are going to use LeakCanary, it is with the intent of availing yourself of its mostly-automatic detection of leaked activities. The recipe for doing that involves calling `install()` on the `LeakCanary` class when your process starts, such as in `onCreate()` of a custom `Application` subclass.

The sample app has such a class, `CanaryApplication`:

```
package com.commonsware.android.async;

import android.app.Application;
import com.squareup.leakcanary.LeakCanary;

public class CanaryApplication extends Application {
  @Override
  public void onCreate() {
```

```
    super.onCreate();

    LeakCanary.install(this);
  }
}
```

(from Leaks/AsyncTask/app/src/main/java/com/commonsware/android/async/CanaryApplication.java)

This `Application` subclass is registered in the manifest, via `android:name` on the `<application>` element:

```
  <application
    android:name=".CanaryApplication"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name=".AsyncDemo"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
```

(from Leaks/AsyncTask/app/src/main/AndroidManifest.xml)

And that is all that you need, for basic integration.

Adding Manual Leak Checks

`LeakCanary.install()` returns a `RefWatcher` object. If all you want to do is use the semi-automatic activity leak detection, you can safely ignore this return value.

However, if you would like to watch for other objects leaking — fragments, domain model objects, threads, etc. — you can hang onto that `RefWatcher` and, where needed, call `watch()` on it to add an object to watch for leaks. Watching for leaks is not terribly expensive but not free, so be judicious in what you are watching.

Testing with LeakCanary

Once you have LeakCanary integrated, you can try out your app and see if it leaks.

Note that the quasi-automatic activity leak detection is based upon the activity lifecycle. LeakCanary considers an activity to be leaked if it is destroyed and there are still unknown strong references to it. This assume that your activity is destroyed in an ordinary fashion. Hence, how you use your app influences what leaks you find. For example, if you terminate the app process (e.g., swipe away the associated task in the overview screen), you will not find out if any live activities were leaked. Where possible, try to use the BACK button to step your way out of the app when testing, to ensure everything gets destroyed and the most leaks can be found.

The Leak Toast

You will not have a problem determining when a leak is suspected, as a very large `Toast`-style window appears advertising the fact:

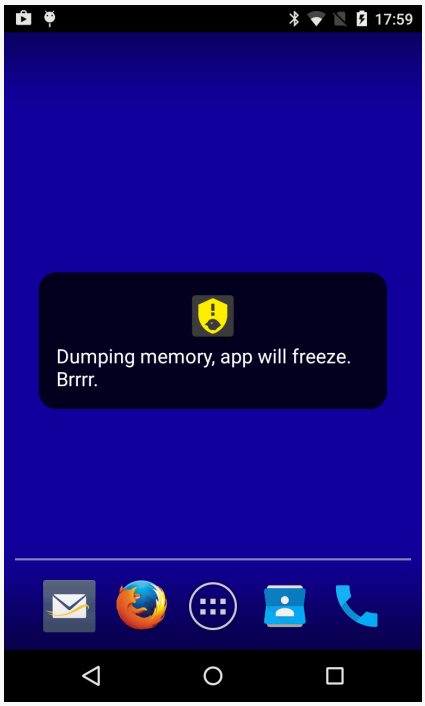


Figure 1039: LeakCanary Heap Dump Window

This is another reason not to watch too many objects, as if you get too many false positives, your productivity will suffer, waiting for all the heap dumps.

Note that it may take a few moments after the activity is destroyed before the message appears, and that it may take a long time after the message disappears before you get final results.

Logcat Output

Whether the leak is "for realz" or a false positive, you will get a report in Logcat:

```
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: In
com.commonsware.android.async:1.0:1.
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * LEAK CAN
BE IGNORED.
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: *
com.commonsware.android.async.AsyncDemo has leaked:
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * GC ROOT
android.view.inputmethod.InputMethodManager$ControlledInputConnectionWrapper.mParentInputMethodManager
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * references
android.view.inputmethod.InputMethodManager.mNextServedView
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * references
android.widget.ListView.mAdapter
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * references
android.widget.ArrayAdapter.mContext
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * leaks
com.commonsware.android.async.AsyncDemo instance
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * Reference
Key: bd3d11b6-2e49-460e-97f9-7b04de398a82
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * Device:
LGE google Nexus 4 occam
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * Android
Version: 5.1.1 API: 22 LeakCanary: 1.3.1
12-21 08:23:31.279 7192-8049/com.commonsware.android.async D/LeakCanary: * Durations:
watch=5174ms, gc=203ms, heap dump=4795ms, analysis=20814ms
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: * Details:
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: * Instance
of android.view.inputmethod.InputMethodManager$ControlledInputConnectionWrapper
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   mActive
= true
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |
mParentInputMethodManager = android.view.inputmethod.InputMethodManager
[id=0x12c289a0]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   mH =
com.android.internal.view.IInputConnectionWrapper$MyHandler [id=0x12cff5e0]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |
mInputConnection = java.lang.ref.WeakReference [id=0x12cff5c0]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |
mMainLooper = android.os.Looper [id=0x12c63de0]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |
mDescriptor = java.lang.String [id=0x70f94d78]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   mObject
= -1201196048
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   mOwner =
```

```
android.view.inputmethod.InputMethodManager$ControlledInputConnectionWrapper
[id=0x12cfe490]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: * Instance
of android.view.inputmethod.InputMethodManager
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   static
$staticOverhead = byte[] [id=0x717d01b1;length=240;size=256]
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   static
CONTROL_START_INITIAL = 256
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   static
CONTROL_WINDOW_FIRST = 4
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   static
CONTROL_WINDOW_IS_TEXT_EDITOR = 2
12-21 08:23:31.280 7192-8049/com.commonsware.android.async D/LeakCanary: |   static
CONTROL_WINDOW_VIEW_HAS_FOCUS = 1
.
.
.
```

(the dump of static and instance fields of `InputMethodManager` goes on for a really long time)

The fact that this just shows up in Logcat (and not via a `Notification`) and that the Logcat dump has "LEAK CAN BE IGNORED" means that this leak is a known Android issue and is not indicative of a leak in your app.

Notification and Activity Output

The `Leaks/StaticWidgetLC` sample project is a clone of the static widget leak scenario from earlier in this chapter. This version has LeakCanary integrated in, though, and LeakCanary catches this leak.

So, after the "Brrr..." window vanishes and you wait several moments for the heap analysis to finish, you will eventually get a `Notification` from LeakCanary. Tapping that shows a "timeline"-style list of objects, starting with a GC root and ending in the leaked object:

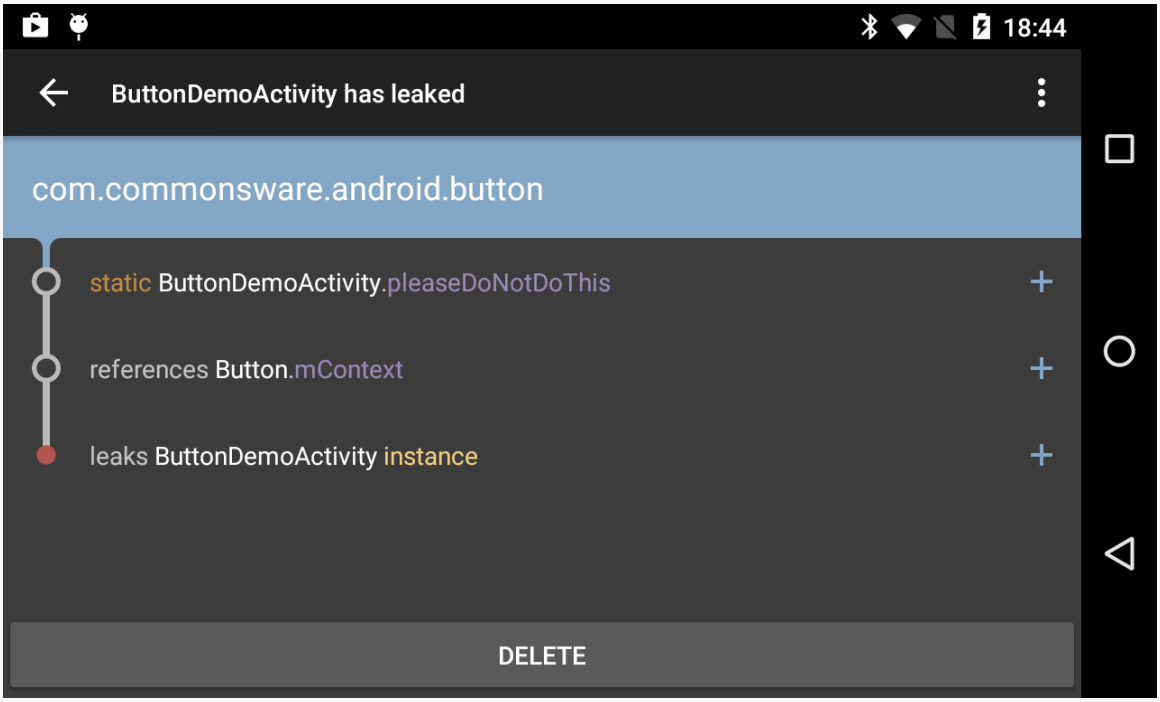


Figure 1040: LeakCanary Diagnostic Activity, As Launched From the Notification

Here, we see that `pleaseDoNotDoThis` holds a reference to the `Button`, which holds a reference to the `Activity`.

This has two advantages over using Android Studio's own leak analysis:

1. It is automatic: we do not have to go and check for leaks ourselves proactively
2. The output can be much easier to read

The "+" icons on the right edge of the rows simply toggle whether the full package name is included in class names:

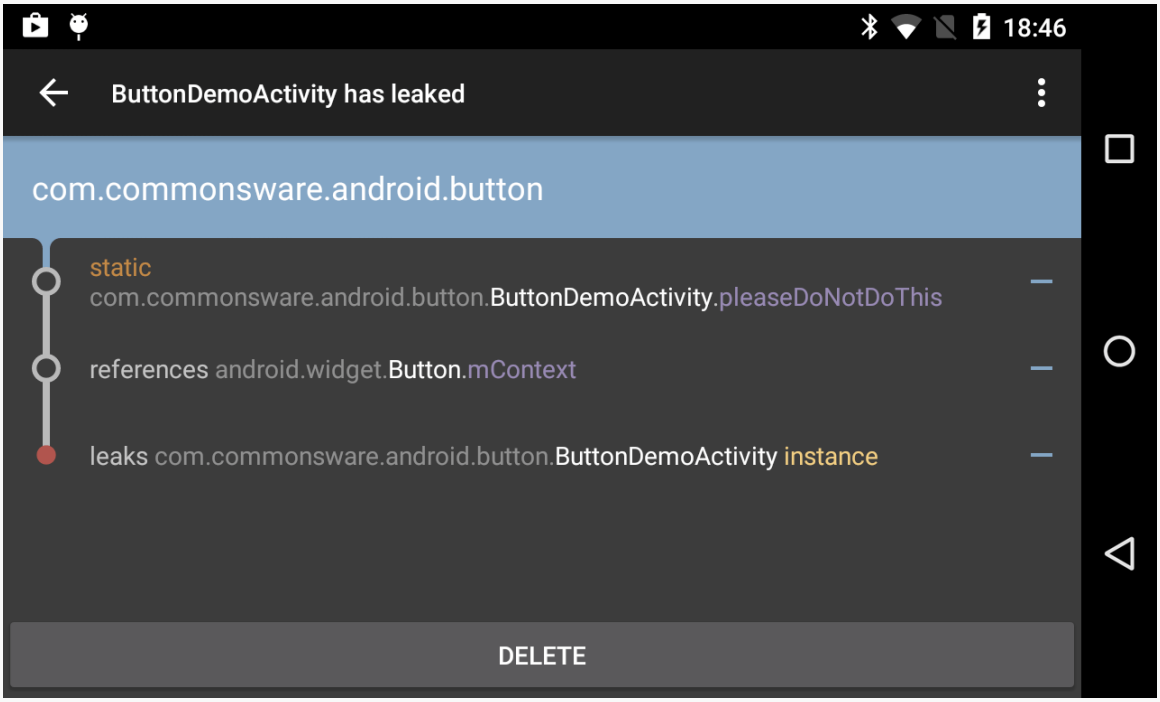


Figure 1041: LeakCanary Diagnostic Activity, Showing Full Package Names

The overflow menu has an option to "Share info", which sends the same information as appears in Logcat to your favorite `ACTION_SEND` implementation (e.g., an email client). "Share heap dump", also in the overflow, forwards the heap dump itself via `ACTION_SEND`, for you to perhaps get over to Android Studio for deeper analysis if that proves necessary.

Pressing the up navigation arrow in the action bar brings up a list of the saved leak reports:

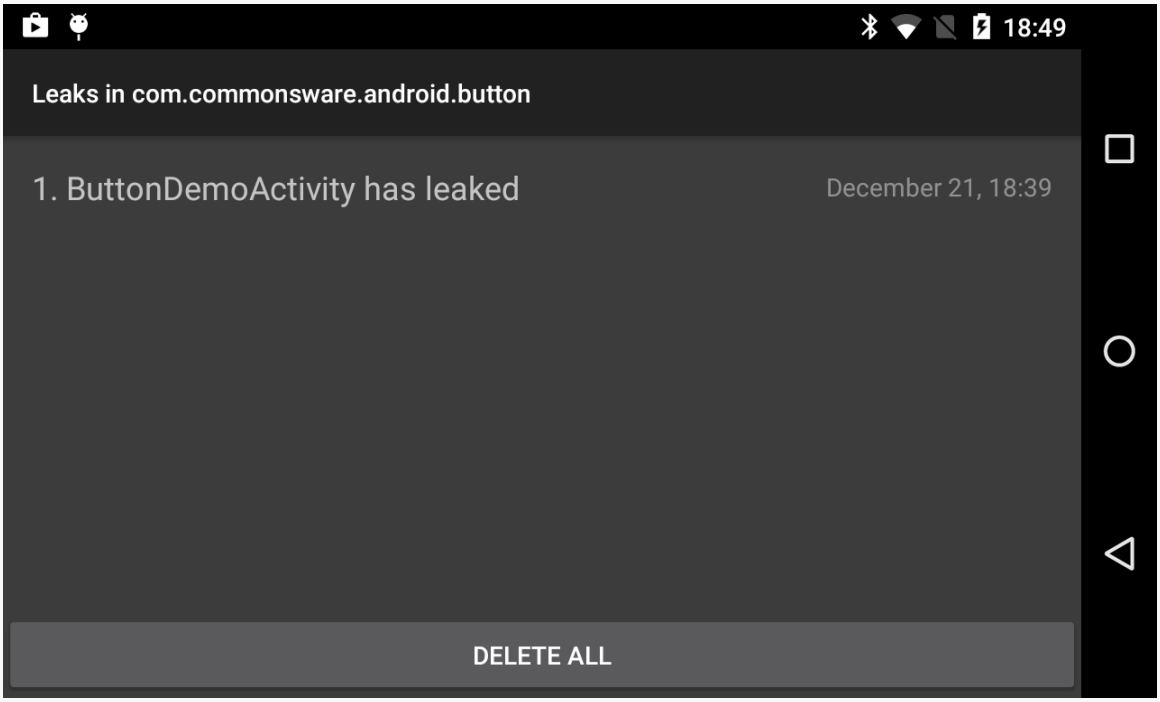


Figure 1042: LeakCanary Report Roster

The "DELETE" button on the diagnostic activity deletes that report; the "DELETE ALL" button on the roster activity deletes all saved reports. By default, LeakCanary saves seven reports and heap dumps, though you can configure this by overriding the `__leak_canary_max_stored_leaks integer` resource with some other value.

The LeakCanary project documentation outlines many other possibilities for tailoring LeakCanary's behavior, including:

- "Whitelisting" certain objects or activity classes, so they do not show up in leak reports
- Sending reports to a server

Issues with System RAM

Your application heap is your little corner of the system RAM on the device that you focus on in your Java development. However, there are other things that you might do that consume system RAM, such as use the NDK to add C/C++ code to your app. How much system RAM you consume overall will have an impact on user acceptance of your app, as the more RAM you use, the more frequently the user's other apps are terminated to make room for you. And, as a result, the more system RAM you use, the more likely it is that *your* process will be terminated when you are not in the foreground, to free up RAM for other apps. Hence, while system RAM is not something you necessarily think about as often as you do your application heap, it *is* something that you should pay attention to, at least a little bit.

This chapter will explain a bit more about the relationship between your app and system RAM, how you can measure how much system RAM your app is consuming, and how you can reduce that consumption.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Android's process model. Reading the chapter on issues with the application heap is also a good idea.

Can't We All Just Get Along?

Alas, we have not invented the device with infinite RAM, nor the application that takes zero memory. In fact, our devices have fairly limited RAM (e.g., 1GB), and our apps therefore fight over that memory. That includes both apps that the user runs

explicitly (e.g., via the home screen launcher) and apps that run based upon external factors (e.g., the app that receives a GCM push event and uses that trigger to update some data).

The good news is that the user tends to be a bit oblivious to all of the comings and goings of apps. Android keeps process around while it can and terminates them as needed to free up system RAM for other processes, without the user's explicit involvement. Of course, power users might try to employ "task managers" and the like to be more involved in decision-making, but that's something they opted into, not something that was forced upon them, the way that older mobile operating systems like Windows Mobile required.

However, there is a fundamental assumption in Android that apps play fair. The per-process heap limits — and the fact that apps do not necessarily have to use all the way up to those limits — means that a given Android device can power many processes at once. That starts to break down when apps do various things to consume an excessive amount of system RAM, more than what the per-process heap limit would normally constrain them to. Hence, it is a good idea to keep tabs on how much you use of system RAM, so that you can be a "good citizen" and not cause the user undue angst or force them to employ task managers to try to keep you in line.

Contributors to System RAM Consumption

There are many factors that contribute towards your system RAM consumption, including:

- Your heap usage, up to the per-process heap limit, for each process that you are running
- Your native libraries (`.so` files) from the NDK
- The system RAM allocated by that native code, which does not count against your per-process heap usage

In addition, the reporting tools usually allocate a portion of shared RAM to your app. Your app's process is forked from the `zygote` process, which contains the Dalvik runtime environment, framework JAR (for all those `android.*` classes), and related libraries. Your app shares that memory with all other processes forked from the `zygote`. However, to reflect the fact that there is this overhead, your app's share of it (roughly calculated as the amount of shared RAM divided by the number of processes) tends to get added to your memory consumption totals.

Measuring System RAM Consumption: Tools

Figuring out how much RAM your application is using is not easy. Or, as Dianne Hackborn put it:

> Note that memory usage on modern operating systems like Linux is an extremely complicated and difficult to understand area. In fact the chances of you actually correctly interpreting whatever numbers you get is extremely low. (Pretty much every time I look at memory usage numbers with other engineers, there is always a long discussion about what they actually mean that only results in a vague conclusion.)

Fortunately, particularly in Android 4.4, a fair bit of work has gone into trying to help us determine how much our apps impact system RAM.

Process Stats in Settings

On Android 4.4, in Settings > Developer Options, you will find:

> Process Stats: Geeky stats about running processes

(here, “geeky” is presumably used as a term of endearment)

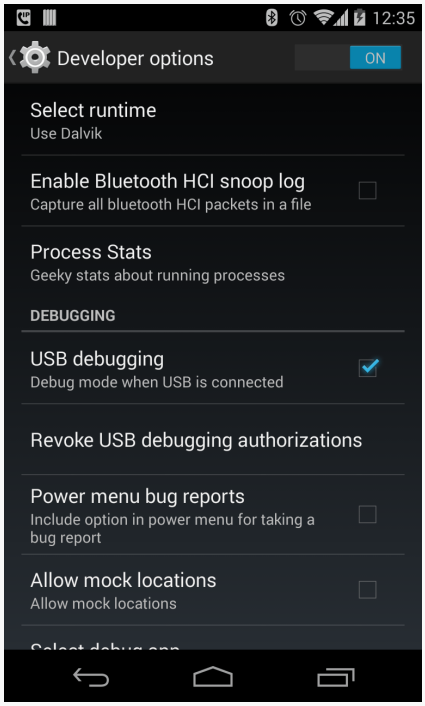


Figure 1043: Developer Options in Android 4.4, Showing "Process Stats"

Tapping on that entry brings up a screen that describes the current state of the system, with respect to RAM:

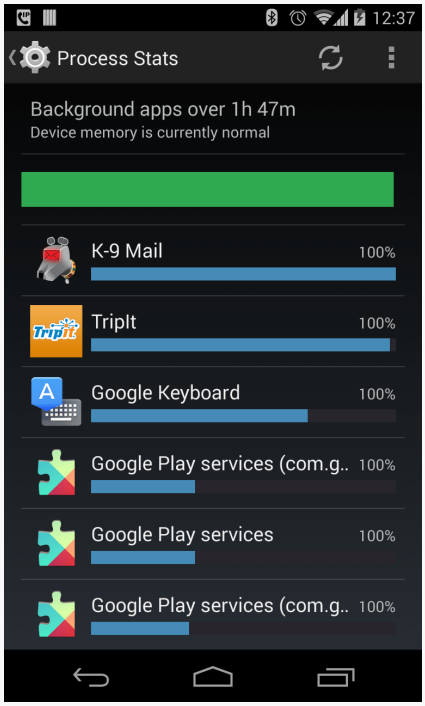


Figure 1044: Process Stats in Android 4.4

While it may look simple, this screen, and its child screens, are remarkably complex, particularly once you start playing around with various options from the action bar overflow.

The Summary

At the top, you will see:

- How much history is being reported in this screen ("1h 47m")
- What the current status of the system is with respect to RAM ("currently normal")
- What the status of the system has been over that range of time, as illustrated by the bar, where green is "normal"

The bar is not so much a timeline as a stacked bar chart, where the mix of red and yellow indicates the amount of time the device was in a low-memory state, contrasted with the green "normal" state.

Usually, your device should be "normal" with a mostly-green or completely-green bar.

The Roster

The list beneath the summary shows some running processes. What is included in this list depends on what mode Process Stats in running in. The summary indicates that our mode is "Background apps". There are three major categories for apps:

1. Foreground, which includes whatever app is truly in the visible foreground, plus any apps that have a foreground service
2. Background, which is pretty much everything else with a service
3. Cached, which are all apps that still have running processes but do not have a service

By default, Process Stats will show background processes.

Each row in the list shows details for a specific process:

- the application label for the app that owns the process (e.g., "K-9 Mail")
- the percentage of time, for the time period Process Stats is reporting, that this process was running (e.g., "100%")
- the "relative computed memory load" of that process, as is indicated by the blue bar, with longer blue bars indicating greater load

The list is sorted in order of "relative computed memory load".

Many background apps will have a running percentage of 100%, indicating that they have an always-running service. Those with a percentage less than 100% indicate an app that had a service running at the point in time of the Process Stats snapshot that you are examining, but did not necessarily have a service running for the entire timeframe (e.g., periodic `IntentService` doing background work)

Refresh and Duration

There is a "refresh" icon in the action bar that will update the current view to reflect changes since you last opened or refreshed the screen.

How long the timeframe is depends a bit upon device operation and also on the "Duration" entry in the overflow menu:

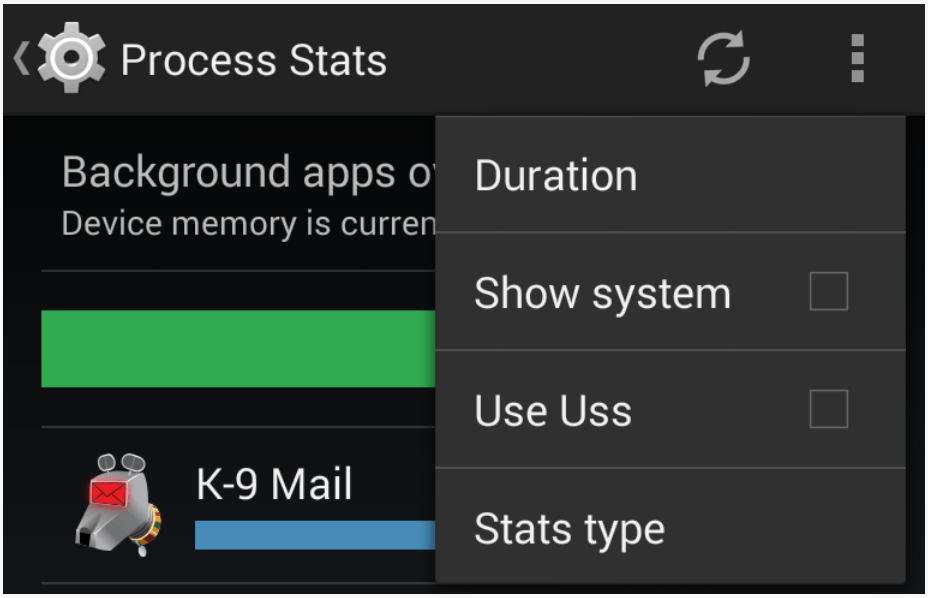


Figure 1045: Process Stats Overflow in Android 4.4

Tapping that gives you a roster of available timeframes:

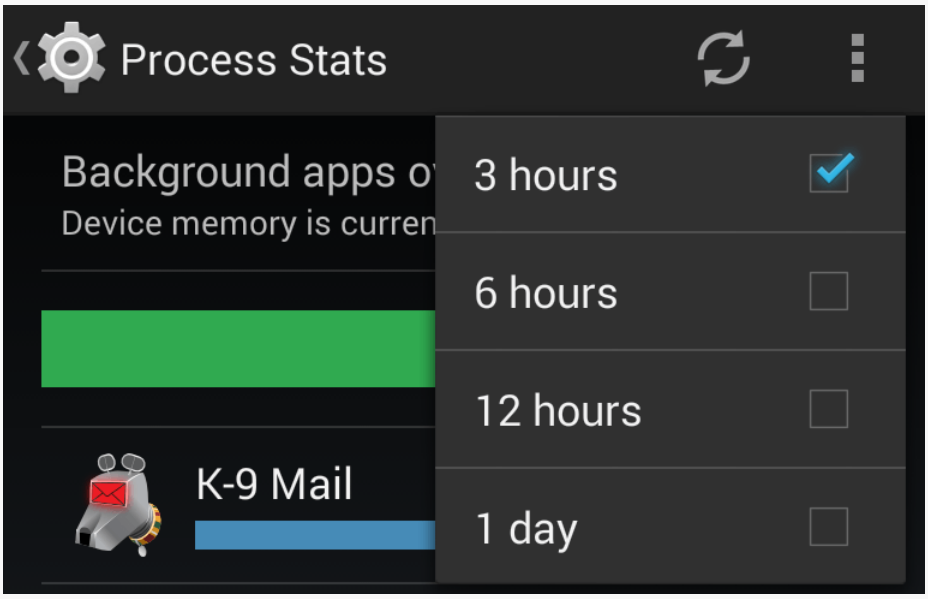


Figure 1046: Process Stats Duration Options in Android 4.4

Even though these items render with checkboxes, they function as radio buttons, so whatever you tap on becomes the new duration. Upon making a change, the summary area will reflect the newly-chosen duration. Note that this choice is not

persistent, as exiting Process Stats via the BACK button and re-entering it returns you to a three-hour duration.

Controlling What is Shown

If your application is not showing up in the background roster, it may be classified as "foreground" (e.g., if you have a foreground service) or "cached" (if not). The "Stats type" overflow option will let you toggle between these categories, to see what processes are reported in each.

Note that an app can appear in more than one roster, since the roster is by *process*. For example, at the time of this writing, Evernote appears in the author's Nexus 4 both in "foreground" and in "cached", for separate processes.

As with the duration, the choice of category is not persistent, and you will be returned to the background process roster if you exit Process Stats via the BACK button and later return to it.

Drilling Down Into an App

Tapping on an item in the list will bring up details about that particular app and process:

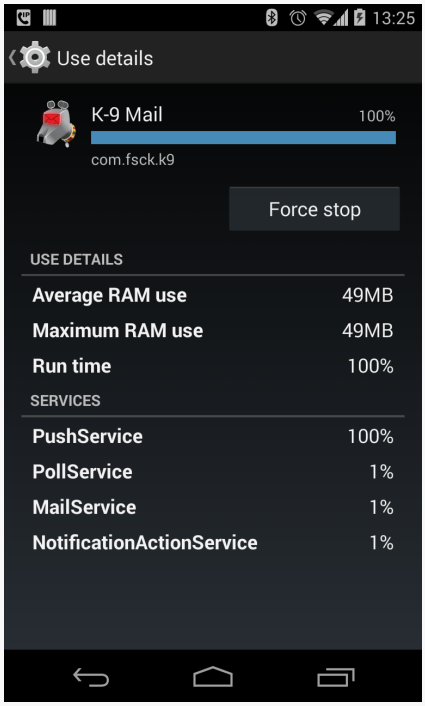


Figure 1047: Process Stats Details for K-9 Mail

The "Average RAM use" value shows how much system RAM is attributed to your app. This will include:

- All system RAM used uniquely by your app (e.g., your heap)
- A portion of system RAM shared by your app and others (e.g., the Dalvik runtime)

This is known as "Proportional Set Size" or "PSS" in Linux, and is a common way of coming up with a simple number for the amount of RAM that a particular process is responsible for.

The blue bar is based on this average RAM use (or PSS) value, multiplied by the percentage of the time that the process was running.

The "Maximum RAM use" value is the highest PSS associated with your process during the period under examination.

Also listed are the services of your app and the percentage of time that they were in a running state. Everlasting services will show up as 100%, while transient services (e.g., `IntentService`) will show up with a much smaller percentage.

How You Want Your App to Appear

Ideally:

- Your app spends most of its time in the "cached" process list, not the "background" or "foreground" list
- Your app has a low percentage of time spent running
- Your app has a low "relative computed memory load" and, ideally, a low "Average RAM use" value

The better you are in these areas, the more likely it is that you are not seriously impacting system RAM.

procstats

The data that powers the Process Stats screen in Settings is also available as human-readable text output, using the `adb shell dumpsys procstats` command, against your Android 4.4 device or emulator.

Running that command will give you three blocks of information:

- Process memory usage, aggregated over the last 24 hours
- Process memory usage, aggregated over the last 3 hours
- A snapshot of the current memory usage, at the time you ran the command

This can be a long report, even for just one of those blocks. For example, here is the last-3-hours block, run on the author's personal Nexus 4:

```
AGGREGATED OVER LAST 3 HOURS:
  * com.android.bluetooth / 1002:
           TOTAL: 100% (7.8MB-7.8MB-7.8MB/7.0MB-7.0MB-7.0MB over 2)
          Imp Fg: 100% (7.8MB-7.8MB-7.8MB/7.0MB-7.0MB-7.0MB over 2)
  * com.csipsimple:sipStack / u0a135:
           TOTAL: 100% (10MB-10MB-10MB/9.2MB-9.2MB-9.2MB over 1)
          Imp Fg: 99% (10MB-10MB-10MB/9.2MB-9.2MB-9.2MB over 1)
         Service: 1.2%
  * system / 1000:
           TOTAL: 100% (56MB-60MB-63MB/51MB-55MB-58MB over 2)
      Persistent: 100% (56MB-60MB-63MB/51MB-55MB-58MB over 2)
  * com.android.nfc / 1027:
           TOTAL: 100% (6.3MB-6.3MB-6.3MB/5.4MB-5.5MB-5.5MB over 2)
      Persistent: 100% (6.3MB-6.3MB-6.3MB/5.4MB-5.5MB-5.5MB over 2)
  * tunein.player.pro / u0a97:
```

```
         TOTAL: 100% (8.2MB-8.2MB-8.2MB/6.9MB-6.9MB-6.9MB over 3)
       Service: 100% (8.2MB-8.2MB-8.2MB/6.9MB-6.9MB-6.9MB over 3)
* android.process.acore / u0a0:
         TOTAL: 100% (15MB-15MB-15MB/14MB-14MB-14MB over 1)
        Imp Fg: 0.00%
       Service: 100% (15MB-15MB-15MB/14MB-14MB-14MB over 1)
* com.google.android.gms / u0a23:
         TOTAL: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
       Service: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
* com.espn.radio:com.urbanairship.process / u0a142:
         TOTAL: 100% (6.6MB-6.6MB-6.6MB/5.3MB-5.3MB-5.3MB over 3)
       Service: 100% (6.6MB-6.6MB-6.6MB/5.3MB-5.3MB-5.3MB over 3)
* com.android.launcher / u0a35:
         TOTAL: 100% (73MB-73MB-73MB/69MB-69MB-69MB over 8)
           Top: 100% (73MB-73MB-73MB/69MB-69MB-69MB over 8)
* com.android.systemui / u0a116:
         TOTAL: 100% (38MB-39MB-41MB/35MB-37MB-38MB over 2)
    Persistent: 100% (38MB-39MB-41MB/35MB-37MB-38MB over 2)
* com.android.phone / 1001:
         TOTAL: 100% (26MB-26MB-26MB/25MB-25MB-25MB over 2)
    Persistent: 100% (26MB-26MB-26MB/25MB-25MB-25MB over 2)
* com.tripit / u0a85:
         TOTAL: 100% (44MB-44MB-44MB/41MB-41MB-41MB over 1)
        Imp Fg: 0.72%
       Service: 99% (44MB-44MB-44MB/41MB-41MB-41MB over 1)
* com.google.process.location / u0a23:
         TOTAL: 100% (17MB-17MB-17MB/14MB-14MB-14MB over 2)
        Imp Fg: 100% (17MB-17MB-17MB/14MB-14MB-14MB over 2)
* com.google.android.inputmethod.latin / u0a34:
         TOTAL: 100% (37MB-37MB-37MB/36MB-36MB-36MB over 2)
        Imp Fg: 100% (37MB-37MB-37MB/36MB-36MB-36MB over 2)
* com.google.process.gapps / u0a23:
         TOTAL: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
       Service: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
* com.fsck.k9 / u0a128:
         TOTAL: 100% (50MB-50MB-50MB/47MB-47MB-47MB over 2)
       Service: 100% (50MB-50MB-50MB/47MB-47MB-47MB over 2)
* com.rememberthemilk.MobileRTM / u0a89:
         TOTAL: 17%
        Imp Fg: 8.6%
       Service: 8.6%
      Receiver: 0.11%
      (Cached): 83% (36MB-36MB-37MB/34MB-35MB-35MB over 9)
* android.process.media / u0a15:
         TOTAL: 8.9% (5.4MB-5.4MB-5.4MB/4.6MB-4.6MB-4.6MB over 1)
       Service: 8.9% (5.4MB-5.4MB-5.4MB/4.6MB-4.6MB-4.6MB over 1)
      Receiver: 0.01%
      (Cached): 91%
```

```
* com.google.android.apps.maps / u0a39:
        TOTAL: 4.2%
      Service: 4.2%
     (Cached): 96% (121MB-121MB-121MB/105MB-105MB-106MB over 4)
* com.google.android.apps.genie.geniewidget / u0a21:
        TOTAL: 3.9% (5.6MB-5.6MB-5.6MB/4.7MB-4.7MB-4.7MB over 1)
      Service: 3.9% (5.6MB-5.6MB-5.6MB/4.7MB-4.7MB-4.7MB over 1)
     Receiver: 0.00%
     (Cached): 96% (5.7MB-5.7MB-5.7MB/4.8MB-4.8MB-4.8MB over 1)
* com.google.android.tts / u0a29:
        TOTAL: 1.7%
      Service: 1.7%
     (Cached): 20% (24MB-24MB-24MB/23MB-23MB-23MB over 2)
* com.evernote / u0a86:
        TOTAL: 0.41%
       Imp Bg: 0.31%
      Service: 0.10%
     Receiver: 0.00%
     (Cached): 100% (15MB-15MB-16MB/14MB-14MB-14MB over 4)
* org.mozilla.firefox / u0a100:
        TOTAL: 0.39%
      Service: 0.39%
     (Cached): 100% (4.4MB-6.2MB-7.4MB/3.5MB-5.2MB-6.4MB over 5)
* com.csipsimple / u0a135:
        TOTAL: 0.18%
       Imp Fg: 0.02%
      Service: 0.14%
     Receiver: 0.02%
     (Cached): 100% (4.2MB-4.2MB-4.2MB/3.2MB-3.2MB-3.2MB over 8)
* com.stackexchange.marvin / u0a154:
        TOTAL: 0.17%
      Service: 0.17%
     Receiver: 0.00%
     (Cached): 100% (32MB-32MB-32MB/30MB-30MB-30MB over 2)
* com.google.android.youtube / u0a67:
        TOTAL: 0.12%
      Service: 0.01%
     Receiver: 0.11%
     (Cached): 8.9% (9.3MB-9.3MB-9.3MB/8.0MB-8.0MB-8.0MB over 1)
* com.guywmustang.silentwidget / u0a78:
        TOTAL: 0.05%
      Service: 0.05%
     Receiver: 0.00%
     (Cached): 100% (3.3MB-3.3MB-3.4MB/2.7MB-2.7MB-2.7MB over 4)
* com.google.android.deskclock / u0a14:
        TOTAL: 0.03%
     Receiver: 0.03%
     (Cached): 100% (4.0MB-4.0MB-4.0MB/3.1MB-3.1MB-3.1MB over 2)
```

```
  * com.google.android.gallery3d / u0a20:
           TOTAL: 0.03%
        Receiver: 0.03%
        (Cached): 9.0% (6.2MB-6.2MB-6.2MB/5.3MB-5.3MB-5.3MB over 1)
  * com.szyk.myheart / u0a130:
        (Cached): 100% (35MB-35MB-35MB/31MB-31MB-31MB over 2)
  * org.wikipedia / u0a98:
        (Cached): 100% (22MB-22MB-22MB/18MB-18MB-18MB over 2)
  * com.android.mms / u0a41:
        (Cached): 100% (23MB-23MB-23MB/21MB-21MB-21MB over 2)
  * nz.co.softwarex.hundredpushupsfree / u0a168:
        (Cached): 100% (23MB-23MB-23MB/20MB-20MB-20MB over 2)
  * com.commonsware.books.android / u0a148:
        (Cached): 100% (18MB-18MB-18MB/15MB-15MB-15MB over 2)
  * com.android.providers.calendar / u0a7:
        (Cached): 100% (3.6MB-3.6MB-3.6MB/2.8MB-2.8MB-2.8MB over 2)
  * com.google.android.calendar / u0a6:
        (Cached): 100% (4.7MB-4.7MB-4.7MB/3.8MB-3.8MB-3.8MB over 2)

Run time Stats:
  SOff/Norm: +12m36s333ms
  SOn /Norm: +1m11s367ms
      TOTAL: +13m47s700ms

          Start time: 2014-04-12 06:01:36
  Total elapsed time: +3h54m32s538ms (partial) libdvm.so chromeview
```

Unfortunately, it is rather cryptic and rather long.

There are various command-line switches you can add to help manage the output. Use the `-h` switch to see the full roster. Some notable options:

- `-csv` switches the output to be in CSV format, for importing into a spreadsheet or running through an analysis tool, with other switches (e.g., `-csv-proc`) to control what is included in the CSV output
- `--current` will only report the current snapshot
- `--hours NNN` will only report the aggregate over the stated number of hours
- `--full-details` provides a somewhat more documented report, at the cost of greatly increasing its verbosity

Also, including a package name (e.g., `com.commonsware.android.sample`) at the end of the command line will constrain the output to solely that package, which is useful if you are only looking to examine your own app's data.

The numbers in parentheses (e.g., `(4.7MB-4.7MB-4.7MB/3.8MB-3.8MB-3.8MB over 2)`) report:

- the minimum proportional set size (PSS) seen
- the average PSS seen
- the maximum PSS seen
- the minimum unique set size (USS) seen, where this is the amount of memory consumed by your app that is *not* shared with other processes (i.e., it is how much memory that would be freed if your process were terminated)
- the average USS seen
- the maximum USS seen
- the number of samples taken of the memory during the timeframe being analyzed

The listing also shows the percentage of time your process was in various states (e.g., cached vs. service vs. "important foreground")

meminfo

Older devices that do not support `procrank` can support `meminfo`, accessed via `adb shell dumpsys meminfo`. Run as-is, it will generate a report of all processes and their PSS, plus the same roster broken down into various process categories (e.g., foreground, cached), and other summary data. The report for the same Nexus 4 that generated the `procrank` shown earlier in this chapter is:

```
Applications Memory Usage (kB):
Uptime: 95955008 Realtime: 788076654

Total PSS by process:
   120505 kB: com.google.android.apps.maps (pid 17490 / activities)
    74627 kB: com.android.launcher (pid 1696 / activities)
    62422 kB: system (pid 1366)
    57757 kB: surfaceflinger (pid 968)
    51706 kB: com.fsck.k9 (pid 2937 / activities)
    44725 kB: com.tripit (pid 2414 / activities)
    41642 kB: com.android.systemui (pid 1498 / activities)
    38546 kB: com.google.android.inputmethod.latin (pid 1635)
    36640 kB: com.rememberthemilk.MobileRTM (pid 12255 / activities)
    35518 kB: com.szyk.myheart (pid 24618 / activities)
    32588 kB: com.stackexchange.marvin (pid 28230 / activities)
    27128 kB: com.android.phone (pid 1667)
    23641 kB: com.android.mms (pid 15197 / activities)
    23236 kB: nz.co.softwarex.hundredpushupsfree (pid 20599 / activities)
```

```
 22483 kB: org.wikipedia (pid 11895 / activities)
 18044 kB: com.commonsware.books.android (pid 12036 / activities)
 16968 kB: com.google.process.location (pid 1775)
 16895 kB: com.google.android.gms (pid 1651)
 16521 kB: com.google.process.gapps (pid 1803)
 15822 kB: com.evernote (pid 26284)
 15219 kB: android.process.acore (pid 11926)
 11336 kB: zygote (pid 969)
 10694 kB: com.csipsimple:sipStack (pid 25539)
  9575 kB: com.google.android.youtube (pid 31932)
  8390 kB: tunein.player.pro (pid 2699)
  7860 kB: com.android.bluetooth (pid 5513)
  7669 kB: org.mozilla.firefox (pid 20839)
  6786 kB: com.espn.radio:com.urbanairship.process (pid 2526)
  6719 kB: mediaserver (pid 971)
  6599 kB: com.google.android.gallery3d (pid 31894)
  6493 kB: com.android.nfc (pid 1681)
  5916 kB: com.google.android.apps.genie.geniewidget (pid 22781)
  5677 kB: android.process.media (pid 25240)
  4308 kB: com.csipsimple (pid 28166)
  4145 kB: com.google.android.deskclock (pid 12379)
  3472 kB: com.guywmustang.silentwidget (pid 14616)
  3349 kB: rild (pid 967)
  2447 kB: drmserver (pid 970)
  1972 kB: ks (pid 585)
  1876 kB: netd (pid 965)
  1282 kB: wpa_supplicant (pid 26091)
  1217 kB: mm-qcamera-daemon (pid 982)
  1116 kB: sdcard (pid 981)
   618 kB: sensors.qcom (pid 979)
   577 kB: netmgrd (pid 976)
   500 kB: vold (pid 163)
   486 kB: bridgemgrd (pid 974)
   476 kB: thermald (pid 977)
   462 kB: keystore (pid 973)
   439 kB: /init (pid 1)
   375 kB: qmuxd (pid 975)
   262 kB: ueventd (pid 139)
   230 kB: dhcpcd (pid 15630)
   214 kB: qseecomd (pid 1022)
   212 kB: adbd (pid 961)
   210 kB: installd (pid 972)
   189 kB: mpdecision (pid 978)
   181 kB: rmt_storage (pid 164)
   176 kB: dumpsys (pid 489)
   169 kB: qcks (pid 165)
   149 kB: debuggerd (pid 966)
   140 kB: healthd (pid 161)
```

```
      135 kB: efsks (pid 569)
      115 kB: servicemanager (pid 162)
      111 kB: qseecomd (pid 986)

Total PSS by OOM adjustment:
    95497 kB: Native
                57757 kB: surfaceflinger (pid 968)
                11336 kB: zygote (pid 969)
                 6719 kB: mediaserver (pid 971)
                 3349 kB: rild (pid 967)
                 2447 kB: drmserver (pid 970)
                 1972 kB: ks (pid 585)
                 1876 kB: netd (pid 965)
                 1282 kB: wpa_supplicant (pid 26091)
                 1217 kB: mm-qcamera-daemon (pid 982)
                 1116 kB: sdcard (pid 981)
                  618 kB: sensors.qcom (pid 979)
                  577 kB: netmgrd (pid 976)
                  500 kB: vold (pid 163)
                  486 kB: bridgemgrd (pid 974)
                  476 kB: thermald (pid 977)
                  462 kB: keystore (pid 973)
                  439 kB: /init (pid 1)
                  375 kB: qmuxd (pid 975)
                  262 kB: ueventd (pid 139)
                  230 kB: dhcpcd (pid 15630)
                  214 kB: qseecomd (pid 1022)
                  212 kB: adbd (pid 961)
                  210 kB: installd (pid 972)
                  189 kB: mpdecision (pid 978)
                  181 kB: rmt_storage (pid 164)
                  176 kB: dumpsys (pid 489)
                  169 kB: qcks (pid 165)
                  149 kB: debuggerd (pid 966)
                  140 kB: healthd (pid 161)
                  135 kB: efsks (pid 569)
                  115 kB: servicemanager (pid 162)
                  111 kB: qseecomd (pid 986)
    62422 kB: System
                62422 kB: system (pid 1366)
    75263 kB: Persistent
                41642 kB: com.android.systemui (pid 1498 / activities)
                27128 kB: com.android.phone (pid 1667)
                 6493 kB: com.android.nfc (pid 1681)
    74627 kB: Foreground
                74627 kB: com.android.launcher (pid 1696 / activities)
    63374 kB: Visible
                38546 kB: com.google.android.inputmethod.latin (pid 1635)
```

```
              16968 kB: com.google.process.location (pid 1775)
               7860 kB: com.android.bluetooth (pid 5513)
    10694 kB: Perceptible
              10694 kB: com.csipsimple:sipStack (pid 25539)
     5677 kB: A Services
               5677 kB: android.process.media (pid 25240)
    51706 kB: Previous
              51706 kB: com.fsck.k9 (pid 2937 / activities)
    76422 kB: B Services
              44725 kB: com.tripit (pid 2414 / activities)
              16521 kB: com.google.process.gapps (pid 1803)
               8390 kB: tunein.player.pro (pid 2699)
               6786 kB: com.espn.radio:com.urbanairship.process (pid 2526)
   402275 kB: Cached
             120505 kB: com.google.android.apps.maps (pid 17490 / activities)
              36640 kB: com.rememberthemilk.MobileRTM (pid 12255 / activities)
              35518 kB: com.szyk.myheart (pid 24618 / activities)
              32588 kB: com.stackexchange.marvin (pid 28230 / activities)
              23641 kB: com.android.mms (pid 15197 / activities)
              23236 kB: nz.co.softwarex.hundredpushupsfree (pid 20599 / activities)
              22483 kB: org.wikipedia (pid 11895 / activities)
              18044 kB: com.commonsware.books.android (pid 12036 / activities)
              16895 kB: com.google.android.gms (pid 1651)
              15822 kB: com.evernote (pid 26284)
              15219 kB: android.process.acore (pid 11926)
               9575 kB: com.google.android.youtube (pid 31932)
               7669 kB: org.mozilla.firefox (pid 20839)
               6599 kB: com.google.android.gallery3d (pid 31894)
               5916 kB: com.google.android.apps.genie.geniewidget (pid 22781)
               4308 kB: com.csipsimple (pid 28166)
               4145 kB: com.google.android.deskclock (pid 12379)
               3472 kB: com.guywmustang.silentwidget (pid 14616)

Total PSS by category:
   200246 kB: Dalvik
   172738 kB: Native
   151232 kB: Graphics
    89516 kB: Dalvik Other
    65741 kB: .dex mmap
    62904 kB: GL
    59426 kB: .so mmap
    58237 kB: Other dev
    24084 kB: Unknown
    15480 kB: .apk mmap
     8404 kB: Stack
     7562 kB: Other mmap
     1112 kB: Ashmem
     1099 kB: .ttf mmap
```

```
     160 kB: .jar mmap
      16 kB: Cursor
       0 kB: code mmap
       0 kB: image mmap
       0 kB: Memtrack

Total RAM: 1878788 kB
 Free RAM: 1258215 kB (402275 cached pss + 625628 cached + 230312 free)
 Used RAM: 780542 kB (515682 used pss + 192112 buffers + 3180 shmem + 69568 slab)
 Lost RAM: -159969 kB
   Tuning: 192 (large 512), oom 122880 kB, restore limit 40960 kB (high-end-gfx)
```

However, if you add a package name to the command (e.g., `adb shell dumpsys meminfo com.commonsware.books.android`), you will get a more detailed report about that specific app:

```
Applications Memory Usage (kB):
Uptime: 96120803 Realtime: 788242449

** MEMINFO in pid 12036 [com.commonsware.books.android] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     7642     7616        0        0     8732     8553      178
  Dalvik Heap     1920     1472        0        0     9860     9819       41
 Dalvik Other     1568     1428        0        0
        Stack      316      316        0        0
      Ashmem       128       68        0        0
    Other dev        4        0        4        0
     .so mmap     2372      528      116        0
    .apk mmap      130        0        8        0
    .ttf mmap       18        0        0        0
    .dex mmap     1232       12      936        0
   Other mmap      198        4        8        0
      Unknown     2516     2516        0        0
        TOTAL    18044    13960     1072        0    18592    18372      219

 Objects
               Views:       48         ViewRootImpl:        1
         AppContexts:        3           Activities:        1
              Assets:        2        AssetManagers:        2
       Local Binders:        8        Proxy Binders:       17
    Death Recipients:        0
     OpenSSL Sockets:        0

 SQL
         MEMORY_USED:       75
```

```
 PAGECACHE_OVERFLOW:        3          MALLOC_SIZE:       62

DATABASES
     pgsz     dbsz   Lookaside(b)          cache  Dbname
        1     6263             17         0/16/1  /data/data/
com.commonsware.books.android/databases/booksearch.db
```

This can show you:

- How much memory is being consumed by your Dalvik bytecode (`.dex mmap`), native libraries used directly by you or by framework components (`.so mmap`), etc.
- How many objects of various types are in the Dalvik heap, such as views and activities
- How much memory is used by SQLite for its page cache and related process-level buffers, plus which databases you have open that are contributing to memory consumption

Note that the combination of the `Private Dirty` and `Private Clean` columns is roughly analogous to the USS reported by `procstats`, in that it represents the amount of memory private to your process and that would be released should your process be terminated.

Measuring System RAM Consumption: Runtime

Some of the same information that the aforementioned reports contain is available at runtime via `ActivityManager` and other framework classes.

getMemoryInfo()

`getMemoryInfo()` on `ActivityManager` will fill in a supplied `ActivityManager.MemoryInfo` object. This will report to you:

- The total memory on the device (`totalMem`)
- The "available memory" (whose definition is a bit unclear) (`availMem`)
- What level of "available memory" is considered "low" and should trigger Android to start terminating processes beyond those that are cached, such as ones with running services (`threshold`)
- Whether we are presently in such a low-memory state (`lowMemory`)

getMyMemoryState()

`ActivityManager` also has a `getMyMemoryState()` method, on API Level 16+, that will populate an `ActivityManager.RunningAppProcessInfo` object with information about your process. While not everything in this object will be filled in, you will be able to get:

- The last trim level reported to your activities and `Application` via `onTrimMemory()` (`lastTrimLevel`)
- What importance level the OS considers your process to be in (`importance`), such as `IMPORTANCE_FOREGROUND` and `IMPORTANCE_EMPTY`
- For processes in the `IMPORTANCE_BACKGROUND` category — meaning the process has outstanding activities but is not in the foreground and has no service — the relative standing of the process compared to other background processes from a least-recently-used standpoint (`lru`), where lower numbers mean more recent usage

getProcessMemoryInfo()

The `getProcessMemoryInfo()` method on `ActivityManager` returns an array of `Debug.MemoryInfo` objects corresponding to the array of `int` process IDs (pids) that you pass in. The `Debug.MemoryInfo` objects report how much memory those identified processes are consuming. Of particular note, `getTotalPss()` returns the PSS for that process.

To get the `Debug.MemoryInfo` for your own process, you can use `getMemoryInfo()` on the `Debug` class, rather than find your own process ID and use `getProcessMemoryInfo()`. Or, on API Level 14+, you can simply call `getPss()` on `Debug` directly to find out your PSS.

Learn To Let Go (Of Your Heap)

Part of the reason for worrying a bit about your system RAM consumption is simply to "play nice" with the other apps that the user wants to use. However, since part of Android's decision-making about what processes to terminate tie into how much RAM those processes take up, the lower your system RAM footprint, the more likely it is that you can hang around for a while.

Part of reducing your system RAM consumption involves cleaning up your heap.

The reason the framework calls callback methods like `onTrimMemory()` is to help you reduce your heap usage to avoid `OutOfMemoryError` exceptions. However, allowing objects to be garbage-collected not only gives you more heap space, but it *also may reduce your system RAM footprint*.

To limit your system RAM usage, your process is not allocated all of its possible heap when the process is started up. Instead, the heap starts small and expands as you allocate more and more memory. However, the reverse is also true: if you release memory, the heap can shrink, returning RAM to the system. Android expands the heap on a "paged" basis, allocating more system RAM to add more pages to the heap. If, as a result of garbage collection, Dalvik sees that there are too many totally empty pages, Dalvik can free up those pages, returning them to the OS for use by other processes.

As noted in the chapter on the application heap, Dalvik's garbage collector is non-compacting, meaning that it does not move objects around to try to clean up pages or otherwise coalesce free memory blocks. Hence, a fragmented heap not only limits how well you can allocate new memory, but it also inhibits Dalvik's ability to reduce your system RAM usage.

Issues with Battery Life

Most Android devices are powered by batteries — Android TV is the biggest class of device that is not. Batteries are wonderful gizmos with one major problem: they are *always* running out of power.

Hence, users are very sensitive to battery consumption. Their ability to use their phones as actual *phones*, let alone for Android apps, depends on having enough battery power. The more apps drain the battery, the more frequently the user has to find a way to recharge the phone, and the more frequently the user fails and their phone shuts down.

The catch is that you may not notice the battery issues in your day-to-day development. The Android emulator's emulated battery does not drain based on you running your app. Your devices are often connected to your development machine via USB for testing and debugging, meaning they are perpetually being charged. Unless you are a regular user of your own app, you might not notice any increased power drain.

This part of the book is focused on helping you understand what is draining power and what you can do to be kinder and gentler on your users' batteries.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

You're Getting Blamed

Users, for better or worse, have limited ability to determine what is responsible for draining the battery of their phone. Their #1 tool for this is the "Power Usage Summary" screen in the Settings app, sometimes referred to as the "battery blame screen".

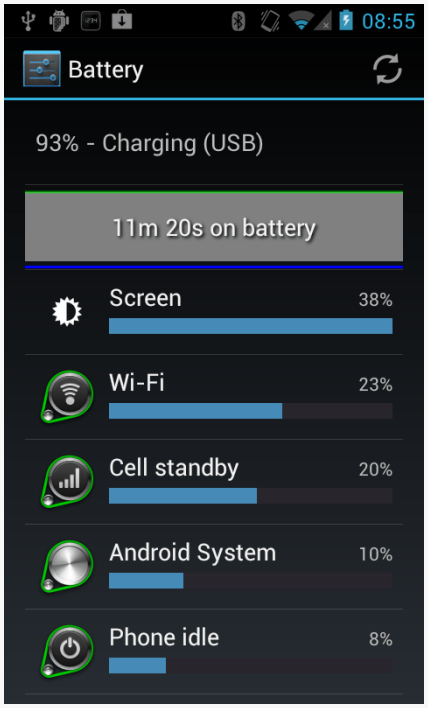


Figure 1048: Battery Screen from Settings App

This lists both device features (e.g., the display) and applications. Android incrementally improves the accuracy of this screen with each passing release, trying to make sure the user understands what specifically is consuming the power.

If your application starts appearing on this screen, and the user does not feel that it is justified, the user is likely to become irritated with you.

Now, your appearance on this list might be perfectly reasonable. If you have written a video player app, and the user has just watched a few hours' worth of video, it is very likely that you will appear on this list and will be justified in your battery consumption.

However, anything that you can do to not appear on this screen, or appear lower in the list, will help with user acceptance of your app.

This part of the book will show you how to measure your power usage and ways of trying to use less of it.

Not All Batteries Are Created Equal

Roughly speaking, battery capacity is proportional to screen size. Larger screens mean physically larger devices, and since the rest of the components (e.g., CPU) tend to be the same size, a larger device offers more room for a larger battery. This is good, as the screen is one of the major power draws on a device, and bigger screens draw more power.

Conversely, the battery on a "wearable" — whether eyewear like Google Glass, a smartwatch, or other form factors — tends to be much smaller than average, just because the wearables are physically smaller. A wearable is likely to have a battery with less than a third of the capacity of a phone, which in turn may have a battery with less than a third of the capacity of a large tablet.

Hence, depending upon where your app will be running, the amount of battery available in total will vary widely. What might be considered acceptable battery consumption on a tablet would be considered excessive on a wearable.

Stretching Out the Last mWh

Sometimes, what the user wants your app to do in one case is not what the user wants your app to do in other cases. Serious power-draining might be reserved for when the device is plugged in, or when the device has at least such-and-so power remaining. The user may value the last milliwatt-hours (mWh) more than others and want your application to use less power in those circumstances.

Hence, if your application polls the Internet, you might offer a feature to poll less frequently, or perhaps not at all, when power is low. If your application uses GPS to find a location (e.g., automatic "check-ins" to social networks like Foursquare), you might offer to skip such actions when the battery is low. You might want to signal to the user when the battery gets low during playback of a video, or during the game they are in. And so on.

This part of the book will help you identify when the battery is low and strategies for making use of that information.

Power Measurement Options

As with any situation where you are trying to reduce your use of some system resource, you need to be able to accurately measure how much you are using that resource. Otherwise, you will have no idea whether your attempts to reduce usage are helping. It is possible that what you think will consume less of the resource actually consumes *more*, because of unanticipated side-effects. And, if nothing else, if the change makes your code more complicated and does not help much with resource consumption, you may be better served sticking with the original, simpler implementation.

So, when it comes to power usage, it helps to know how much power you are consuming, to determine if your attempts to use less power actually do help.

Unfortunately, compared to things like RAM and bandwidth, power measurement is a significant challenge. You really need to have hardware specifically instrumented to report power consumption for pieces of that hardware (CPU versus screen versus GPS versus mobile data radio versus ...). Even if you cannot get power usage per component, just having accurate power consumption overall is not something you can necessarily get from any Android device. Alas, getting that level of power usage knowledge can be troublesome in its own right, for a variety of reasons.

This chapter will explore a few ways of measuring power usage, along with the pros and cons of that approach.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

batterystats and the Battery Historian

Android 5.0 brought us "Project Volta", an initiative to reduce the amount of power consumed by apps, the framework classes, and the OS itself.

Part of what we got from Project Volta is `batterystats`, a dump of data pertaining to power consumption. Since that data dump can be large and inscrutable, we also have the Battery Historian, a tool that can convert key `batterystats` output into a timeline of events.

However, none of this is especially well-documented at this time, and so the usefulness of these utilities is limited at present. The following sections provide some basic guidance for trying to use these tools.

Running a Test

First, since `batterystats` is obtained via `adb shell dumpsys`, you will want `adb` to be in your `PATH`, by adding your SDK installation's `platform-tools/` directory to your `PATH` environment variable.

Then, in a terminal, run:

```
adb shell dumpsys batterystats --enable full-wake-history
adb shell dumpsys batterystats --reset
```

This will ensure that `batterystats` captures all relevant information about `WakeLock` behavior, and it resets all of the logs.

At this point, run your tests. Ideally, you would do so in a fairly power-neutral environment, such as not using a USB cable for an `adb` connection (as that charges the device).

When your test scenario is complete, run `adb shell dumpsys batterystats`, redirecting the output to some file:

```
adb shell dumpsys batterystats > /tmp/bs.txt
```

You can optionally supply your `applicationId` as part of the `batterystats` command, which will restrict the output to events pertaining to your app. However, some events that are from other processes, like the Play Services Framework, may be

of interest to you. You will need to experiment to determine which mode (full or filtered for your app) will work best for you.

Interpreting the Text Output

Depending on how long your test runs, the information included in the `batterystats` output can be anywhere from tens of KB to tens of MB in size. The output will also vary by Android OS release.

Battery History

The file will lead off with the "Battery History" section:

```
Battery History (13% used, 35KB used of 256KB, 68 strings using 4232):
          0 (9) RESET:TIME: 2014-10-25-19-54-07
          0 (2) 100 status=not-charging health=good plug=none temp=207 volt=4296
+running +wake_lock +sensor +phone_scanning +audio +screen phone_state=out
+wifi_running +wifi wifi_signal_strength=4 wifi_suppl=completed
proc=u0a3:"android.process.acore"
          0 (2) 100 proc=u0a29:"com.android.calendar"
          0 (2) 100 proc=1027:"com.android.nfc:sendui"
          0 (2) 100 proc=u0a7:"com.google.android.gms"
          0 (2) 100 proc=1000:"WebViewLoader-armeabi-v7a"
          0 (2) 100 proc=u0a32:"com.google.android.configupdater"
          0 (2) 100 proc=u0a7:"com.google.process.location"
          0 (2) 100 proc=u0a12:"com.android.launcher"
          0 (2) 100 proc=1001:"com.android.server.telecom"
          0 (2) 100 proc=u0a7:"com.google.process.gapps"
          0 (2) 100 proc=u0a55:"com.nuance.xt9.input"
          0 (2) 100 proc=u0a33:"com.google.android.deskclock"
          0 (2) 100 proc=u0a5:"android.process.media"
          0 (2) 100 proc=u0a20:"com.android.systemui"
          0 (2) 100 proc=1027:"com.android.nfc"
          0 (2) 100 proc=1001:"com.android.phone"
          0 (2) 100 proc=u0a37:"com.google.android.gallery3d"
          0 (2) 100 proc=u0a58:"com.commonsware.android.job"
          0 (2) 100 proc=u0a2:"com.android.providers.calendar"
          0 (2) 100 proc=u0a16:"com.android.vending"
          0 (2) 100 proc=u0a7:"com.google.android.gms.unstable"
          0 (2) 100 proc=u0a42:"com.google.android.inputmethod.latin"
          0 (2) 100 proc=u0a11:"com.google.android.partnersetup"
          0 (2) 100 top=u0a58:"com.commonsware.android.job"
          0 (2) 100 wake_lock_in=-1:"screen"
          0 (2) 100 user=0:"0"
          0 (2) 100 userfg=0:"0"
```

```
     +36ms (2) 100 +wake_lock_in=u0a7:"Wakeful StateMachine: GeofencerStateMachine"
     +37ms (2) 100 -wake_lock_in=u0a7:"Wakeful StateMachine: GeofencerStateMachine"
    +104ms (2) 100 +wake_lock_in=u0a7:"UlrDispatchingService"
    +146ms (2) 100 +wake_lock_in=u0a7:"GCoreFlp"
    +147ms (2) 100 -wake_lock_in=u0a7:"GCoreFlp"
    +150ms (2) 100 -wake_lock_in=u0a7:"UlrDispatchingService"
  +2s001ms (2) 100 volt=4243
  +6s779ms (3) 100 -sensor +wake_lock_in=1000:"ActivityManager-Sleep"
  +6s781ms (2) 100 +wake_lock_in=u0a20:"show keyguard"
  +6s811ms (2) 100 -wake_lock_in=1000:"ActivityManager-Sleep"
  +6s820ms (2) 100 +wake_lock_in=1000:"WifiSuspend"
  +6s835ms (2) 100 -wake_lock_in=1000:"WifiSuspend"
  +6s868ms (2) 100 +wake_lock_in=1013:"AudioMix"
  +6s875ms (2) 100 -wake_lock_in=u0a20:"show keyguard"
  +6s878ms (2) 100 -wake_lock_in=1013:"AudioMix"
  +6s878ms (2) 100 +wake_lock_in=u0a20:"AudioMix"
  +6s941ms (2) 100 +wake_lock_in=1027:"NfcService:mRoutingWakeLock"
  +6s941ms (2) 100 +wake_lock_in=u0a7:"Wakeful StateMachine: GeofencerStateMachine"
  +6s942ms (2) 100 -wake_lock_in=u0a7:"Wakeful StateMachine: GeofencerStateMachine"
  +6s943ms (2) 100 +wake_lock_in=u0a7:"GCoreFlp"
  +6s946ms (2) 100 -wake_lock_in=u0a7:"GCoreFlp"
  +6s968ms (2) 100 -wake_lock_in=1027:"NfcService:mRoutingWakeLock"
.
.
.
```

This contains information about how much the battery was drained during the test run, along with a detailed roster of the power-related events that occurred during the test run. The timestamps on those roster entries are relative to the first entry in the roster. Beyond that, there is little explanation of what the roster entries mean.

Per-PID Stats

Next, there will be a short stanza labeled “Per-PID Stats” and, possibly, “Discharge step durations”:

```
Per-PID Stats:
  PID 0 wake time: +134ms
  PID 536 wake time: +1m35s993ms
  PID 0 wake time: +10s842ms
  PID 881 wake time: +301ms
  PID 536 wake time: +70ms
  PID 989 wake time: +23s167ms
  PID 1136 wake time: +2s974ms
  PID 1193 wake time: +230ms
  PID 0 wake time: +1s123ms
```

```
  PID 617 wake time: +187ms
  PID 536 wake time: +18ms
  PID 536 wake time: +13ms
  PID 627 wake time: +586ms
  PID 536 wake time: +184ms
  PID 3690 wake time: +9m42s965ms

Discharge step durations:
  #0: +4h8m36s976ms to 97 (screen-off, power-save-off)
  #1: +3h7m47s132ms to 98 (screen-off, power-save-off)
```

If you determine your process' PID, you will see how long the process' "wake time" was. The precise definition of "wake time" is undocumented.

Daily Stats

Next may be a section entitled "Daily stats":

```
Daily stats:
  Current start time: 2015-12-12-05-27-33
  Next min deadline: 2015-12-13-01-00-00
  Next max deadline: 2015-12-13-03-00-00
    Package changes:
      Update com.google.android.dialer vers=20312
      Update com.google.android.apps.cloudprint vers=113
      Update com.android.chrome vers=252608301
      Update com.google.android.marvin.talkback vers=40400003
      Update com.google.android.contacts vers=10307
      Update com.commonsware.empublite vers=1
      Update com.commonsware.android.picasso vers=1
  Daily from 2015-12-11-05-55-34 to 2015-12-12-05-27-33:
    Discharge step durations:
      #0: +4h49m4s38ms to 93 (screen-off, power-save-off, device-idle-on)
      #1: +3h53m22s969ms to 94 (screen-off, power-save-off)
      #2: +5h8m4s10ms to 95 (screen-off, power-save-off, device-idle-on)
      #3: +3h33m53s102ms to 96 (screen-off, power-save-off, device-idle-on)
        Discharge total time: 18d 3h 10m 2s 900ms  (from 4 steps)
        Discharge screen off time: 18d 3h 10m 2s 900ms  (from 4 steps)
        Discharge screen off device idle time: 18d 18h 33m 58s 300ms  (from 3 steps)
    Package changes:
      Update com.commonsware.empublite vers=1
  Daily from 2015-12-10-04-55-48 to 2015-12-11-05-55-34:
    Discharge step durations:
      #0: +5h1m15s618ms to 98 (screen-off, power-save-off, device-idle-on)
      #1: +42m34s7ms to 98 (screen-off, power-save-off, device-idle-off)
        Discharge total time: 11d 22h 31m 21s 200ms  (from 2 steps)
```

```
        Discharge screen off time: 11d 22h 31m 21s 200ms  (from 2 steps)
        Discharge screen off device idle time: 20d 22h 6m 1s 800ms  (from 1 steps)
    Package changes:
      Update com.commonsware.ct3 vers=1
      Update com.commonsware.ct3 vers=1
      Update com.commonsware.ct3 vers=1
      Update com.commonsware.ct3 vers=1
      Update com.commonsware.ct3 vers=1
      Update com.commonsware.ct3 vers=1
      Update com.commonsware.android.fsendermnc vers=1
...
```

This indicates, for various time slices, what apps were updated and what the "discharge step durations" are (which is undocumented).

"Statistics since last charge" Summary

Next up will be a "Statistics since last charge" header, with a few summary blocks of data:

```
Statistics since last charge:
  System starts: 0, currently on battery: false
  Time on battery: 11h 30m 29s 177ms (99.9%) realtime, 16m 13s 306ms (2.3%) uptime
  Time on battery screen off: 11h 30m 7s 940ms (99.9%) realtime, 15m 52s 69ms (2.3%)
uptime
  Total run time: 11h 30m 55s 300ms realtime, 16m 39s 430ms uptime
  Start clock time: 2014-10-25-19-54-07
  Screen on: 21s 237ms (0.1%) 2x, Interactive: 20s 191ms (0.0%)
  Screen brightnesses:
    dark 21s 237ms (100.0%)
  Total partial wakelock time: 10m 22s 877ms
  Mobile total received: 0B, sent: 0B (packets received 0, sent 0)
  Phone signal levels:
    none 11h 30m 29s 177ms (100.0%) 0x
  Signal scanning time: 9s 0ms
  Radio types:
    none 11h 30m 29s 177ms (100.0%) 0x
  Mobile radio active time: 0ms (0.0%) 0x
  Wi-Fi total received: 0B, sent: 0B (packets received 0, sent 0)
  Wifi on: 11h 30m 29s 177ms (100.0%), Wifi running: 11h 30m 29s 177ms (100.0%)
  Wifi states: (no activity)
  Wifi supplicant states:
    group-handshake 16ms (0.0%) 12x
    completed 11h 30m 29s 161ms (100.0%) 12x
  Wifi signal levels:
    level(4) 11h 30m 29s 177ms (100.0%) 1x
```

```
Bluetooth on: 0ms (0.0%)
Bluetooth states: (no activity)

Device battery use since last full charge
  Amount discharged (lower bound): 2
  Amount discharged (upper bound): 3
  Amount discharged while screen on: 0
  Amount discharged while screen off: 3

Estimated power use (mAh):
  Capacity: 3448, Computed drain: 107, actual drain: 69.0-103
  Idle: 40.3
  Wifi: 36.4
  Uid u0a58: 14.8
  Uid 0: 12.8
  Uid 1000: 1.75
  Uid u0a20: 0.521
  Uid u0a7: 0.389
  Screen: 0.375
  Uid 1013: 0.0775
  Uid 1001: 0.0219
  Uid u0a42: 0.0168
  Uid u0a12: 0.0138
  Uid 1027: 0.0110
  Uid u0a5: 0.00866
  Uid u0a33: 0.00253
  Uid u0a16: 0.000867
  Uid u0a3: 0.000815
  Uid u0a29: 0.000523
  Uid u0a2: 0.000474
  Over-counted: 4.04
```

The first block has useful data about how much various radios were on, how much data they transmitted, how long the screen was on, how long the device had an outstanding partial `WakeLock`, etc.

Also, the "Estimated power use (mAh)" block is basically the data that underlies the "battery blame screen" in Settings. You will see how many milliamp-hours (mAh) were attributed to your process.

WakeLock Summary

Next up may a summary of `WakeLock` events:

```
All kernel wake locks:
Kernel Wake lock PowerManagerService.WakeLocks: 10m 23s 46ms (717 times) realtime
Kernel Wake lock qcom_rx_wakelock: 9m 9s 836ms (1537 times) realtime
Kernel Wake lock alarm_rtc: 55s 327ms (717 times) realtime
Kernel Wake lock sns_async_ev_wakelock: 16s 525ms (9 times) realtime
Kernel Wake lock power-supply: 15s 730ms (1448 times) realtime
Kernel Wake lock event0-536: 11s 822ms (1484 times) realtime
Kernel Wake lock event2-536: 10s 686ms (1509 times) realtime
Kernel Wake lock event4-536: 10s 234ms (1509 times) realtime
Kernel Wake lock alarm: 7s 972ms (1239 times) realtime
Kernel Wake lock wlan: 1s 608ms (2 times) realtime
Kernel Wake lock PowerManagerService.Display: 611ms (2 times) realtime
Kernel Wake lock main: 608ms (0 times) realtime
Kernel Wake lock KeyEvents: 136ms (1548 times) realtime
Kernel Wake lock mmc0_detect: 38ms (1507 times) realtime
Kernel Wake lock deleted_wake_locks: 28ms (170 times) realtime
Kernel Wake lock event5-536: 20ms (2 times) realtime

All partial wake locks:
Wake lock u0a58 wake:com.commonsware.android.job/.DemoScheduledService: 9m 0s 153ms
(671 times) realtime
Wake lock 1000 *alarm*: 44s 114ms (683 times) realtime
Wake lock u0a7 Checkin Service: 16s 572ms (4 times) realtime
Wake lock 1000 NetworkStats: 7s 347ms (338 times) realtime
Wake lock 1013 AudioMix: 5s 833ms (2 times) realtime
Wake lock 1000 DHCP: 4s 953ms (12 times) realtime
Wake lock u0a7 *net_scheduler*: 1s 504ms (82 times) realtime
Wake lock u0a7 Event Log Service: 1s 98ms (18 times) realtime
Wake lock u0a42 DownloadManager: 322ms (1 times) realtime
Wake lock u0a7 Config Service fetch: 235ms (1 times) realtime
Wake lock u0a7 Icing: 197ms (5 times) realtime
Wake lock u0a7 Event Log Handoff: 143ms (18 times) realtime
Wake lock u0a58 *alarm*: 120ms (24 times) realtime
Wake lock u0a7 GCM_CONN: 61ms (36 times) realtime
Wake lock u0a7 GmsDownloadService: 47ms (1 times) realtime
Wake lock u0a7 *alarm*: 42ms (10 times) realtime
Wake lock u0a7 Wakeful StateMachine: GeofencerStateMachine: 35ms (8 times) realtime
Wake lock 1000 SyncManagerHandleSyncAlarm: 34ms (6 times) realtime
Wake lock u0a7 GCM_HB_ALARM: 34ms (36 times) realtime
Wake lock u0a7 Checkin Handoff: 12ms (4 times) realtime
Wake lock 1000 SyncLoopWakeLock: 8ms (4 times) realtime
Wake lock u0a33 *alarm*: 7ms (4 times) realtime
Wake lock u0a42 *alarm*: 4ms (3 times) realtime
Wake lock u0a7 GCoreFlp: 2ms (5 times) realtime
```

If you do not have a separate section for these, they may be interleaved in the "Statistics since last charge:" data.

Your code will tend to show up in the "All partial wake locks" section, showing how many `WakeLock`s you acquired and for how long overall.

And, if you show up here, you can definitely find out your app's PID — for example, `u0a58` is associated with the `com.commonsware.android.job` package.

Per PID Summary

Next up may be summaries of information per process; otherwise, this information is interleaved in the "Statistics since last charge:" section. Your process will show up somewhere in the list:

```
.
.
.
u0a58:
  Wake lock wake:com.commonsware.android.job/.DemoScheduledService: 9m 0s 153ms
partial (671 times) realtime
  Wake lock *alarm*: 120ms partial (24 times) realtime
  TOTAL wake: 9m 0s 273ms partial realtime
  Foreground activities: 7s 817ms realtime (1 times)
  Foreground for: 12s 959ms
  Active for: 3h 58m 47s 853ms
  Running for: 11h 30m 29s 177ms
  Proc com.commonsware.android.job:
    CPU: 1m 4s 370ms usr + 21s 140ms krn ; 400ms fg
  Proc *wakelock*:
    CPU: 43s 690ms usr + 1m 19s 590ms krn ; 0ms fg
  Apk com.commonsware.android.job:
    672 wakeup alarms
    Service com.commonsware.android.job.DemoScheduledService:
      Created for: 11m 14s 174ms uptime
      Starts: 647, launches: 647
.
.
.
```

As usual, the exact definitions of the information here is largely undocumented.

Installing the Battery Historian

While `batterystats` is part of the Android 5.0+ runtime environment, and tools like `adb` are part of the Android SDK, the Battery Historian is neither. Instead, it is a

separate project that you have to download to your development machine from its GitHub project.

The original implementation of the Battery Historian was a Python script. This is still available as the `historian.py` file in that GitHub repository, until such time as Google elects to delete it.

Battery Historian 2.0 is now a server written in the Go programming language. This requires a fair bit more work to set up, as you need to:

- install a Go compiler
- download Go dependencies manually
- modify your `PATH` environment variable, plus add new environment variables
- deal with the intrinsic hassles and risks of running an unnecessary server

This chapter focuses on the original Python script.

Running the Battery Historian

Once you have downloaded that Python script, and assuming that you have a Python interpreter installed, you can run the script, supplying it with the output of your `batterystats` run, and redirecting the script's output to an HTML file:

```
python historian.py /tmp/bs.txt > /tmp/bs-report.html
```

Interpreting the Historian Output

You can then load that HTML into a Web browser (Chrome-flavored ones are probably a good choice, given that it is Google-generated HTML). This will give you a timeline across the horizontal axis, with event categories culled from the "Battery History" section of the `batterystats` output on the vertical axis:

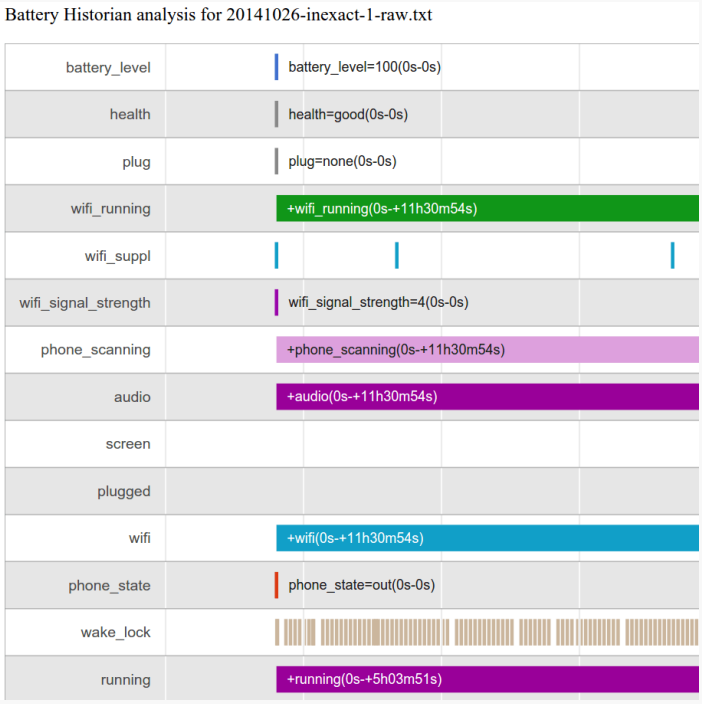


Figure 1049: Battery Historian Timeline, Partial View

Figure 1050: Battery Historian Timeline, Additional

The length of the bar shows the approximate duration of the event, though really short events have a *de minimus* length to give you something to see. Hovering your mouse over one of the bars brings up a pop-up with more details about that event:

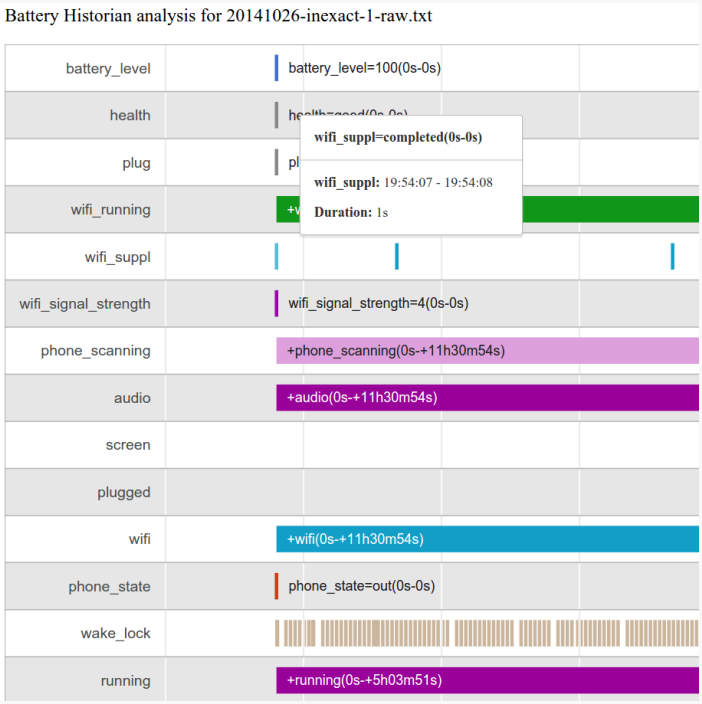


Figure 1051: Battery Historian Timeline, Partial View, with Pop-Up

If the bar is really long, you may need to scroll your browser horizontally to see the pop-up, as the rendering of the pop-up location does not seem to pay attention to the browser viewport very well.

Below the main chart is a "Zoom" field that you can use to change the scale of the horizontal axis, along with an "Event summary":

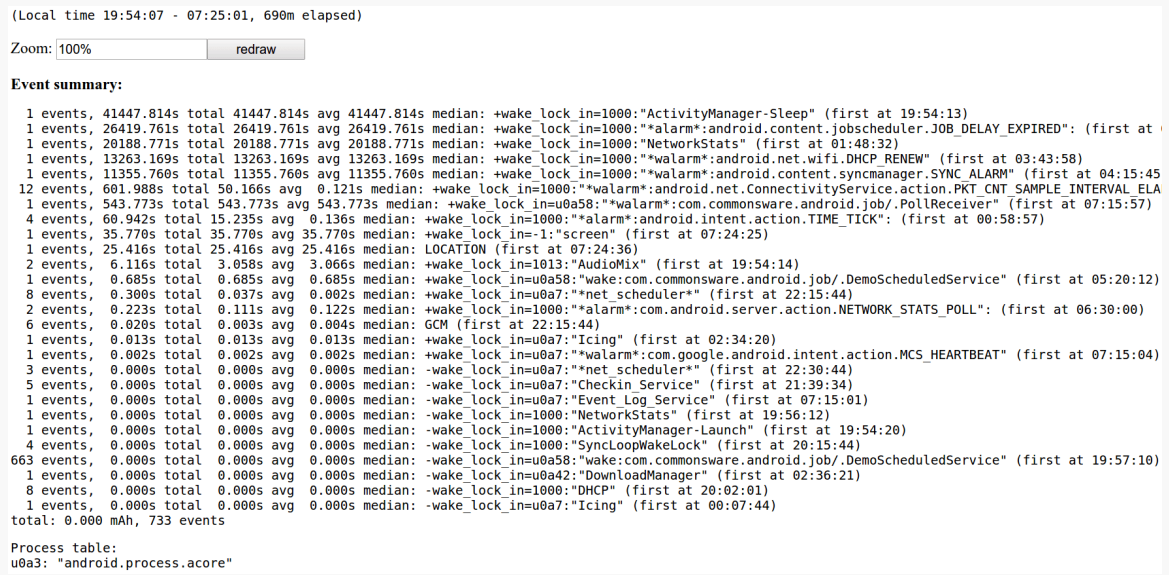


Figure 1052: Battery Historian Timeline, Zoom and "Event summary"

Again, this is largely undocumented.

PowerTutor

Perhaps the best-known third-party power analyzer is PowerTutor. PowerTutor is the outcome of a research project from the University of Michigan, with a bit of assistance from Google. In principle, PowerTutor is capable of letting you know power consumption on a device, much along the lines of what Trepn can record on a Qualcomm MDP. In practice, PowerTutor is significantly less powerful and sophisticated.

PowerTutor was created with the HTC Dream (T-Mobile G1), HTC Magic (T-Mobile G2), and Nexus One in mind. Its power output values will be as accurate as they could make it for those devices. If you run PowerTutor on other hardware, the results will be less accurate.

You can obtain PowerTutor from the Play Store, or from the PowerTutor Web site, or you can compile it from source.

PowerTutor is not tied to testing a particular application. As such, you can simply run PowerTutor whenever you want from its launcher icon, then press "Start Power Profiler" in the main activity:

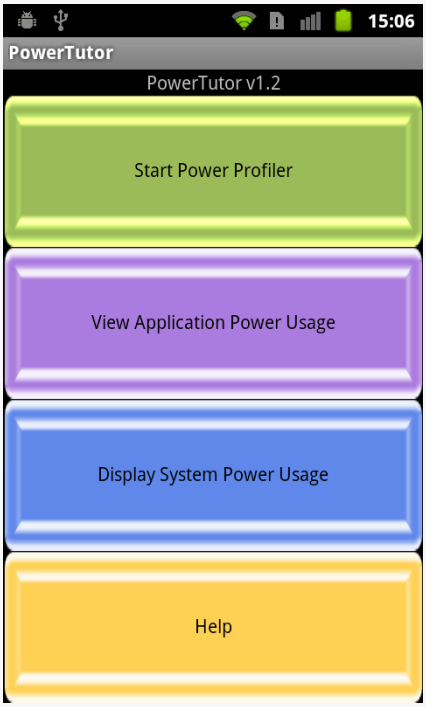


Figure 1053: The PowerTutor main activity

At this point, you can start playing with your application, or running your unit test suite, or whatever. When you want to get an idea of how much power you have been consuming, you can switch back to the PowerTutor activity and choose "View Application Power Usage". This brings up a list of processes and toggle buttons to show various power consumption values for each:

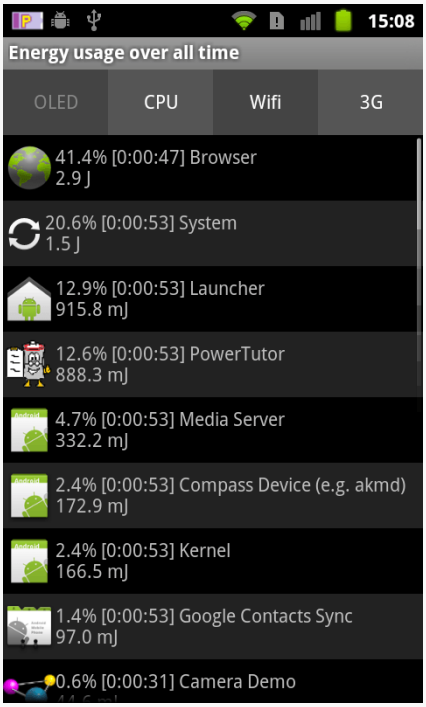


Figure 1054: The PowerTutor application roster

Tapping the list entry brings up a graph for that particular process, though since this information is only available while PowerTutor is recording new data, the graph is usually empty unless you have logic running in the background:

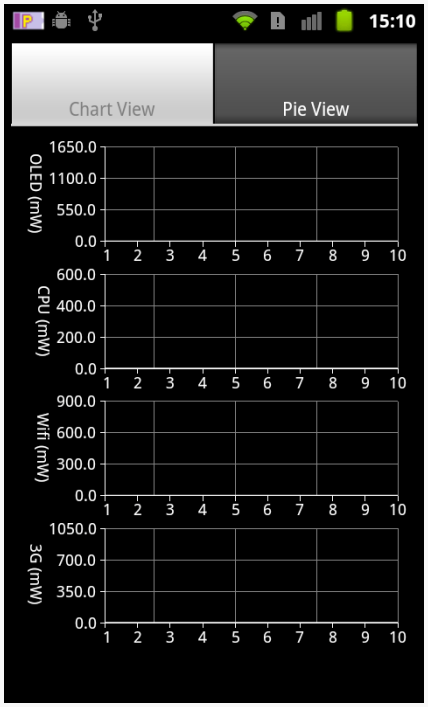


Figure 1055: The PowerTutor live charts for a single process current power consumption

You can also bring up a chart showing what portion of your power consumption came from various sources for the whole device, such as a pie chart of current consumption:

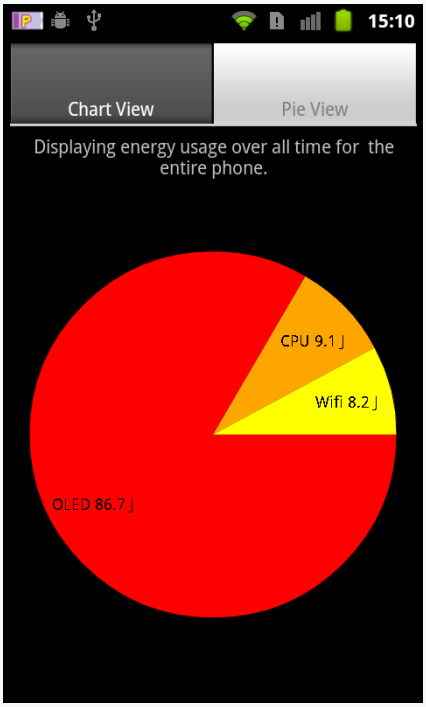


Figure 1056: The PowerTutor pie chart for current overall power consumption

Given that the source code is available, one might augment PowerTutor to:

1. Saving results, both as data files for offline analysis (akin to Trepn's CSV files) or for viewing charts and tables on the device when data is not being actively collected
2. Allowing one to record application states, akin to Trepn, to better correlate application functionality to saved power results

Battery Screen in Settings Application

Of course, what developers tend to focus on most with power is the battery consumption screen in the Settings application, as shown in a previous chapter:

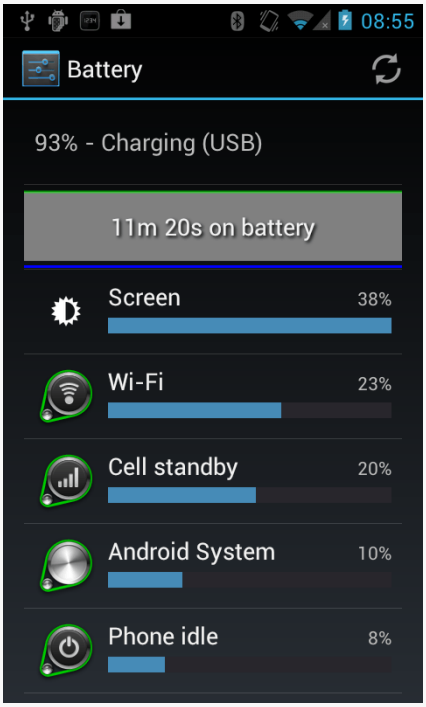


Figure 1057: Battery Screen from Settings App

After all, this is what users will tend to focus on — anything showing up in here is a source of blame for whatever power woes the user believes she is experiencing. Conversely, if your application does not show up in this screen during normal operation, then there is no compelling reason for you to do further analysis, as users will tend to be oblivious to your actual power consumption.

If you do show up in the list, tapping on your entry can give you some more details of what power you consumed and why:

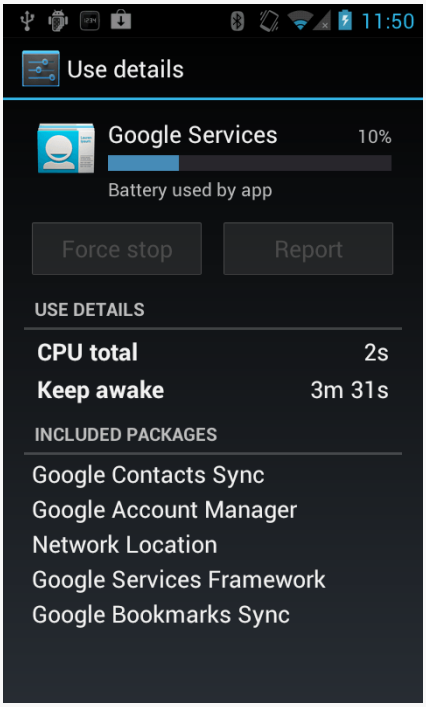


Figure 1058: Battery Details Screen from Settings App

However, the information contained in here is mostly guesswork, using a more refined version of the same approach that PowerTutor uses. Ordinary Android hardware simply lacks enough fine-grained power measurement instrumentation to do an accurate job of apportioning power usage among different processes. So, the details of how long you kept the CPU powered on may be accurate, but the percentage of battery consumption associated with your app is just an estimate.

BatteryInfo Dump

Yet another possibility on older Android devices is to use the `adb shell dumpsys batteryinfo` command from your command prompt or terminal on your development workstation. This will emit a fair amount of data that probably means something to somebody, such as general device information:

```
Battery History:
       -1h00m56s463ms 096 20030002 status=discharging health=good
plug=none temp=191 volt=4060 +screen +wake_lock +sensor
brightness=medium
       -1h00m52s490ms 096 22030302 +wifi phone_state=off
       -1h00m51s844ms 096 2703d102 +phone_scanning +wifi_running
```

```
phone_state=out data_conn=other
       -1h00m49s303ms 096 2743d102 +wifi_scan_lock
         -57m48s766ms 095 2743d102
         -53m24s627ms 095 2743d100 brightness=dark
         -53m17s620ms 095 0741d100 -screen -wake_lock
         -53m17s107ms 095 0740d100 -sensor
         -38m17s007ms 095 0642d100 -wifi_running +wake_lock
         -38m08s998ms 095 0640d100 -wake_lock
            -54s781ms 095 4640d100 status=full plug=usb temp=193
volt=4084 +plugged

Per-PID Stats:
  PID 96 wake time: +12s75ms
  PID 177 wake time: +1s13ms
  PID 458 wake time: +1s898ms
  PID 326 wake time: +3s925ms
  PID 205 wake time: +2s107ms
  PID 415 wake time: +843ms
  PID 96 wake time: +281ms

Statistics since last charge:
  System starts: 0, currently on battery: false
  Time on battery: 1h 0m 1s 682ms (0.3%) realtime, 8m 21s 883ms
(0.0%) uptime
  Total run time: 16d 11h 13m 34s 654ms realtime, 2h 9m 37s 404ms
uptime,
  Screen on: 7m 37s 868ms (12.7%), Input events: 0, Active phone
call: 0ms (0.0%)
  Screen brightnesses: dark 7s 7ms (1.5%), medium 7m 30s 861ms (98.5%)
  Kernel Wake lock "SMD_DS": 2s 368ms  (3 times) realtime
  Kernel Wake lock "mmc_delayed_work": 1s 210ms  (1 times) realtime
  Kernel Wake lock "SMD_RPCCALL": 56ms  (435 times) realtime
  Kernel Wake lock "power-supply": 575ms  (4 times) realtime
  Kernel Wake lock "radio-interface": 3s 1ms  (3 times) realtime
  Kernel Wake lock "ApmCommandThread": 4ms  (10 times) realtime
  Kernel Wake lock "ds2784-battery": 2s 6ms  (21 times) realtime
  Kernel Wake lock "msmfb_idle_lock": 14ms  (2273 times) realtime
  Kernel Wake lock "kgsl": 51s 482ms  (613 times) realtime
  Kernel Wake lock "rpc_read": 164ms  (272 times) realtime
  Kernel Wake lock "main": 7m 39s 708ms  (0 times) realtime
  Total received: 0B, Total sent: 0B
  Total full wakelock time: 149ms , Total partial waklock time: 31s
14ms
  Signal levels: none 59m 57s 63ms (99.9%) 1x
  Signal scanning time: 59m 57s 63ms
  Radio types: none 641ms (0.0%) 1x, other 59m 56s 973ms (99.9%) 1x
  Radio data uptime when unplugged: 0 ms
  Wifi on: 59m 57s 709ms (99.9%), Wifi running: 22m 35s 424ms
```

```
(37.6%), Bluetooth on: 0ms (0.0%)

  Device battery use since last full charge
    Amount discharged (lower bound): 0
    Amount discharged (upper bound): 1
    Amount discharged while screen on: 1
    Amount discharged while screen off: 0

(... and lots more...)
```

and per-process information (here, showing power used by PowerTutor itself):

```
#10058:
    Wake lock window: 5s 71ms window (1 times) realtime
    Proc edu.umich.PowerTutor:
      CPU: 11s 750ms usr + 4s 530ms krn
      1 proc starts
    Apk edu.umich.PowerTutor:
      Service edu.umich.PowerTutor.service.UMLoggerService:
        Created for: 4m 4s 750ms  uptime
        Starts: 1, launches: 1
```

In principle, one might create tools that use this output — or perhaps steal a peek at the data used by the Settings application – to create something a bit more developer-friendly.

Sources of Power Drain

If you can measure power drain well yourself, that is the best way for you to determine precisely where your power consumption is going. Alas, for various reasons, you may not be able to get good power consumption data.

Which means you may have to guess.

We know the general sorts of things that consume power in a device, such as the screen and the CPU. We know that if we use these things less, we will use less power. Eventually, though, we have an app that does nothing, and while this may result in optimal power usage, we are still likely to get poor reviews, because *the app does nothing*.

What we need is some rough idea of how bad certain things are, so we can weigh our use of those system components appropriately.

This chapter will try to give you some "rule of thumb" heuristics of how to estimate power usage of various system components, plus some general recommendations of how to use less of that particular component without necessarily eliminating useful functionality from your app.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Also note that:

- mA = milliamps, where the ampere (or "amp") is the SI unit of current

- mAH = milliamp-hours, which is how battery capacities are measured (e.g., 2000mAH can power a 200mA draw for 10 hours)

Screen

Screen size and battery size generally trend together. Tablets have bigger batteries and bigger screens than do phones, which in turn are bigger in both areas than are wearables.

A rough rule of thumb is to expect to consume ~10% of the device's battery for every hour you keep the screen on. Or, to look at it another way, on a phone-sized screen, expect a power draw of ~100-200mA, depending on variations in screen size and display technology (e.g., AMOLED).

Normally, the user is in control over how long your app is in the foreground and therefore is "to blame" for the screen being on. There are a couple of cases where you can make the screen be more of a problem.

The first is if you `acquire()` a `WakeLock` (other than a `PARTIAL_WAKE_LOCK`)... and forget to ever `release()` it. Since the `WakeLock` will keep the screen on, the screen will stay on, even if your app is in the background, until such time as your process is terminated or the device shuts down due to low battery.

In fact, such `WakeLock` types have been deprecated, with the last of them being flagged as deprecated in API Level 17. The recommended alternative is to use `android:keepScreenOn` or `setKeepScreenOn()` on some `View`. This will keep the screen on, so long as the activity hosting that `View` is in the foreground. That way, just moving to the background releases the underlying `WakeLock`, allowing the device to return to sleep.

However, in some cases, even that may be insufficient. Suppose that the user is in your activity, and they get distracted, putting down their device for an extended period. Unless you somehow detect the inactivity, and manually turn off the keep-screen-on mode, the screen will stay on indefinitely, until the power is drained. Hence, if you have a decent way of determining if the user is still using your activity, consider using that as a way to determine when the device is inactive (e.g., a `postDelayed()` that gets canceled and rescheduled when the user does something, so if the `postDelayed()` `Runnable` gets invoked, you know the user has done nothing for the delay period). Then, if you know the device is inactive, call `setKeepScreenOn(false)` to return the screen to its normal operating mode.

The academic paper "How is Energy Consumed in Smartphone Display Applications?" has a more extended analysis of screen power draw.

Disk I/O

Disk I/O gets more efficient with bigger operations.

You can see this in something like SQLite, where wrapping a bunch of `INSERT` statements into a single transaction can have substantial benefits in terms of how long the I/O takes.

Not surprisingly, this has a similar impact on power consumption:

- Writing 1GB of data 1,000 bytes at a time is about twice as expensive as is writing it 10,000,000 bytes at a time
- Writing 1GB of data 100 bytes at a time is about five times as expensive as is writing it 1,000 bytes at a time

Hence, you want to try to batch up your disk I/O, where possible, to do fewer, bigger operations, rather than lots of little ones. This includes:

- Batching database I/O in a transaction, as noted above
- Caching data that you intend to log to disk in memory and only writing when your in-memory buffer reaches a certain size or age (though beware the dangers of your process being terminated before you get a chance to write the data)
- Consider using larger buffer sizes with `BufferedInputStream` and `BufferedOutputStream`, if you can afford the heap space, though the 8KB defaults are not that bad

As a rough model, consider disk I/O to draw ~200mA. The smaller the I/O operations, the more time it takes you to accomplish the work, and hence the less efficient those operations are.

While disk I/O is relatively expensive while it is occurring, most apps are not continuously reading or writing, and therefore the total impact to the battery will not be that bad. Apps that *do* continuously use the disk — such as music or video players — will consume quite a bit of power.

WiFi and Mobile Data

Internet access via WiFi and mobile data networks is another area that you, the developer, tend to control. Some apps require continuous Internet access and only while in the foreground, like a streaming media player. But many more apps wind up doing Internet access periodically in the background, looking for new information on some server somewhere. Unfortunately, these are the sorts of "vampire" apps that can drain the battery without users necessarily being aware of it. Individually, these apps might not even appear all that bad, but when a device has dozens of them, the combined impact results in poor battery life.

Moreover, we also have the problem of dealing with multiple ways of getting to the Internet. Simple solutions will leave us totally oblivious to the differences in downloading via WiFi versus mobile data, at the potential cost in battery consumption. Slightly less-simple solutions optimize for mobile data, to try to minimize power drain in that model. More-elaborate solutions detect what sort of connection we have (using `ConnectivityManager`) and choose among different strategies as connectivity changes.

Here are some things you can do to try to help manage your Internet power consumption.

Use Less

The simplest, rough-cut way to consume less power for Internet access is to do less Internet access in the first place. The less time you spend downloading (or uploading) data, the less power you tend to draw while doing so. In a very coarse approximation, battery consumption will be proportional to bandwidth consumption.

And, of course, consuming less bandwidth can have other benefits, particularly for people on metered mobile data plans.

There are chapters elsewhere in the book that cover ways to deal with bandwidth consumption for bandwidth's sake.

Use What You Already Downloaded

For data that is likely to be unchanging, use a disk cache, so you can avoid downloading the same content again. Such a cache can be used at two levels:

1. Simply by having the file in the cache can be a signal to your app that you already have the data and can avoid any sort of request to fetch it again.
2. For HTTP, by recording some additional details (`If-Modified-Since` and `ETag` headers), you can make a request to the server to download the content again, where the server can tell you if you already have the current copy of the content (via a `304` response code).

Many of the Internet libraries discussed earlier in this book offer disk caching as part of their services.

Use In Batches

As noted earlier in this section, in a very coarse approximation, battery consumption will be proportional to bandwidth consumption.

Unfortunately, that approximation is pretty coarse.

We as developers tend to think of Internet access as being like a faucet with two states: on and off. In reality, wireless radios tend to have three states: full power, low power, and standby mode. Opening a socket will bring the radio to full power. An idle radio (no packets transferred) will drop to low power after a while, and eventually back to standby mode. Not surprisingly, the power draw for full power is substantially more than low power, which in turn is more than standby.

However, this model introduces some problems:

- There is some latency to move from standby or low power to full power. This slows down data transfer while the radio "warms up".
- The idle time needed to transition to a lower power state is substantial, with values in the 5-15 second range well within reason. This means that making a request has lingering power cost even *after* our request has completed.

The net is that you want to bring the radio to full power as few times as possible (to minimize the percentage of time we are slowly dropping back to standby and consuming power while we do). And, while we are at full power, we want to do all necessary — or perhaps possibly necessary — data transfers, to avoid having to go back to full power again any time soon.

In other words, you want to batch your network I/O. This is reminiscent of the recommendations to batch disk I/O from earlier in this chapter.

So, for example, if you are going to upload data to a server, use that same pulse of work to download anything that needs downloading, rather than having separate schedules for uploads and downloads. Doing more in a batch and having fewer batches will reduce the cost of the power state changes.

Use When the Server Wants You To

One common pattern for Internet access is to poll a server. This is fairly easy to code, using something like `AlarmManager` to get control every so often.

However, this approach resembles children in the back seat of a car, frequently pestering their parents with "Are we there yet?".

Just as the parents will tell the children "We will get there when we get there, and we will tell you when we get there", you can take a similar approach, using Google Cloud Messaging (GCM). Rather than poll the server periodically, have the server contact your app on the device when there is data ready to be downloaded. This works well in cases where polls are likely to result in "yes, we have no data" responses — the pushes can be far less frequent than the polls would be. This can also reduce load on your servers, for not having to respond to poll requests across all your users.

Note, though, that the battery benefits are from using GCM itself. From the standpoint of an app, GCM is "always on", and the power consumed by GCM is attributed to Android itself, not to the app. Hence, pushes are almost "free" from the standpoint of power cost. This will *not* be the case if you "roll your own" push system (MQTT, WebSockets, etc.). In this case, you are attempting to keep a long-lived socket yourself, *in addition* to the one maintained by GCM. Clearly, there are ways to do this that minimize the power consumption of the long-lived socket connection, but that is not easy to accomplish. Hence, you need to weigh the costs of depending upon the Play Services SDK and routing your communications through Google's servers with the costs of trying to do your own separate push mechanism in a battery-friendly fashion.

Use When Android Wants You To

If server push through GCM is impractical (e.g., you do not control the server), you can reduce your power use for Internet access by batching *across apps*, in addition to batching within your app.

What Google wants you to use for synchronizing data with a server is the `SyncManager`. This is an overly-complicated framework that, among other things,

gives you control to sync to the server at the same time that other apps needing to sync get control. That way, we can "warm up" the wireless radio once and handle several apps' worth of data transfers at once. `SyncManager` will be covered in this book eventually.

Part of the reason why Android moved to make alarms with `AlarmManager` more "inexact" in API Level 19+ is for this same sort of batching. While `AlarmManager` certainly can be used for a variety of purposes, a lot of apps use it for Internet data transfer. Allowing Android to control when those alarms occur allows Android to try to coalesce them, and perhaps even time them to happen when `SyncManager`-led transfers occur, with the objective of minimizing the number of times we bring the wireless radio out of standby mode.

Use Additional Reading

The Android developer documentation has a series of "training" pages on minimizing power consumption for data transfers. This expands upon Reto Meier's Google I|O presentations that touch upon this topic.

GPS

In light testing, GPS seems to draw ~35mA. Additional power will be consumed for using those results, though, and so the net effect on the battery will be somewhat higher, depending upon what your app does when it gets a GPS fix.

That figure is corroborated by the academic paper "An Analysis of Power Consumption in a Smartphone", though that paper tested rather old devices (HTC Dream and Nexus One).

Again, different devices will have different components, and some devices' GPS modules may be more or less efficient.

Hence, GPS itself is a power drain, but not a massive one... if what you are doing with the GPS fixes itself is efficient. Keeping the GPS on for several hours will certainly take a chunk out of the battery charge, but if you are doing lots of work (e.g., navigation app) in response to those fixes, several hours may be more than the battery can handle.

If you can get by with the dependency on the Play Services SDK, using `LocationClient` can help here, particularly in cases where the user may not be

moving much, as Google's fused location provider uses the accelerometer to help determine how much they need to use GPS versus other possible means of determining location.

Camera

The camera will consume power while it is actively receiving input, whether that is for the preview frames or for taking full-resolution pictures or video. Of course, it will also consume additional power when recording images to disk, whether those be still photos or continuous video.

A rough guide is that a camera preview will draw ~200mA *plus* the power for screen, CPU, etc. That could easily total over 350mA, even if you are not doing much. Normally, though, the camera preview is on for short periods of time, and only under user control.

A corresponding value for recording video, including the disk I/O and camera preview, would be ~600mA (plus the screen). That is the sort of thing you only want to do in short bursts, as a couple of hours of video recording can really take a bite out of battery. However, once again, normally the user is the one controlling when video is recorded.

Additional Sources

The above sources of power drain are comparatively easy to model and provide a heuristic for determining your possible power usage.

However, there are plenty of other things that can drain the battery, for which this chapter does not provide such a heuristic. In many cases, the usage patterns of the system component will vary so widely that a simple heuristic is unrealistic. In some cases, the power drain from components from different manufacturers will be very different. In some cases, the author of this book simply lacks sufficient expertise with the technology to provide much help (e.g., Bluetooth).

The sections that follow will try to provide some help, though.

CPU/GPU

Perhaps the biggest source of power drain beyond the components listed above will be the processors: the CPU and the GPU. These draw a fair bit of power, which is why processor manufacturers go to great lengths to try to adapt to varying conditions, turning off cores or switching clock speeds, to try to minimize the power drain.

Usually, so long as we are in the foreground, any CPU/GPU usage impact on power will be considered "normal" by the user. Of course, trying to boost performance here can benefit the user, not only in terms of possibly reduced power consumption, but less lag or other forms of sluggishness. Hence, trying to optimize processor utilization is worthwhile.

However, the bigger complaints from the user will come from power drain while your app is in the background. The biggest source of those complaints will come from your use of `WakeLocks`, preventing the device from going into a low-power sleep state.

There are some apps available on the Play Store that reportedly can give you some idea of how long you may be holding a `WakeLock`, however they generally require root, particularly for Android 4.4+.

Sensors

Sensors, more so than many other device components, seem to get sourced from a wide range of manufacturers. They also seem to be tied into the devices differently from device to device. For example, some devices allow sensors to continue collecting data while the device is otherwise in a sleep mode, while many do not.

As such, it is difficult to give much guidance in terms of power drain tied to your use of sensors.

That being said, here are a few notes that may help:

1. Generally speaking, the more you use a sensor, the more likely it is that it will reflect in power drain. However, only some of that power drain will be from the sensor hardware itself. Your application code processing sensor events will bear much of the blame. Reducing the periods of time when you are registered for sensor events, using longer delays between events, and

sensor event batching are ways that you can reduce the power drain associated with the sensors and your associated code.

2. Conversely, in some environments, use of a particular sensor may be "free", insofar as the device uses the sensor itself on a continuous basis. For example, the accelerometer and/or gyroscope is used by devices to detect orientation changes. Hence, those sensors must be powered on regularly, and therefore you cannot be "blamed" for the fact that the sensors are drawing power. Your use of the sensor data may contribute to power drain, of course.

Audio Input and Output

Playing audio through the earpiece, speaker, wired headset, or Bluetooth, will consume some amount of power. The amount will vary by how long you are playing the audio and how the audio is played (e.g., Bluetooth may require more power than on-device audio output). However, in both cases, usually the user has control over the audio, particularly if it is to be playing for a lengthy period of time (e.g., music player), and so the power drain associated with audio playback is less likely to be considered to be a problem, as users will get annoyed with *uncontrolled* power drain, more so than power drain that they can manage themselves.

Recording audio via the on-board microphone or Bluetooth should also consume some incremental power. In cases where the user is in control over when recording is happening, the power drain is unlikely to cause the user much distress.

Where both playback and recording of audio may cause a perceived power problem is in places where the user has less control. For example, an alarm clock app should have some sort of timeout to stop playing the ringtone (or whatever) after some period, if the user fails to respond to the alarm. After all, it is possible that the user is not where the device is and is not in position to stop the alarm. In this case, the power drain will be from several components, audio playback being just one, but it is the uncontrolled nature of the power drain that can get you in trouble.

Addressing Application Size Issues

Sometimes, our apps are just too big, where "too big" can be defined as:

- Bigger than the 100MB limit imposed by the Play Store
- Bigger than some other limit imposed by some other distribution channel
- Big enough that we worry about bandwidth costs, particularly for users on metered data plans
- Big enough that we hit some internal Dalvik limitations

This chapter will review various techniques for trying to keep the size of your app down to a reasonable level.

Prerequisites

This chapter assumes that you have read the core chapters of the book.

The APK Analyzer

One way to reduce the size of your APK is to see what is inside of it that you can get rid of. This is also useful to see if your attempts to get rid of it actually work and reduce the size of the APK file.

Android Studio, starting with version 2.2, has an APK analyzer that will help you do this.

To analyze an APK, choose Build > Analyze APK... from the Android Studio main menu. Unfortunately, this is not aware of your actual APKs from your project, instead dumping you in a standard file-open dialog. You will need to rummage

around your development machine to try to track down the APK to analyze. Look in your module's `build/outputs/apk/` directory, though you may need to specifically build the APK for the build variant that you want to look at.

The analyzer will then generate a tree-table, showing you what things in your app are consuming space:

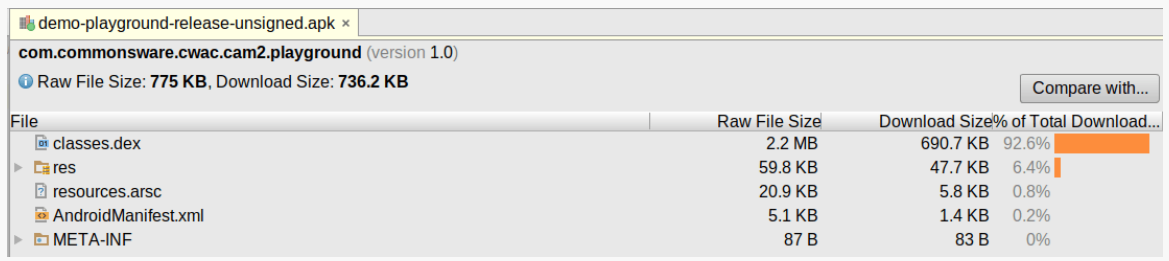


Figure 1059: APK Analyzer, As Initially Launched

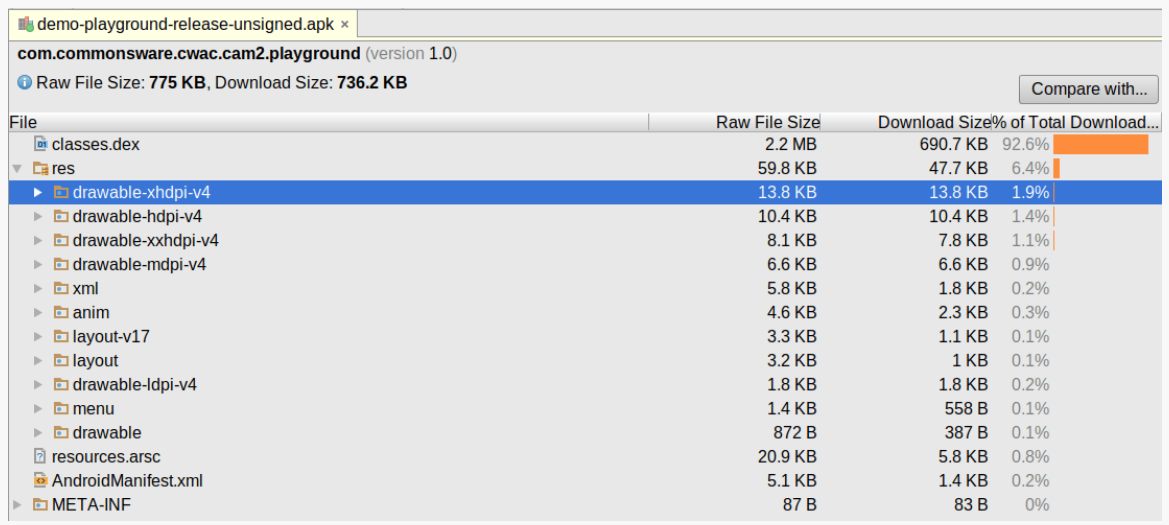


Figure 1060: APK Analyzer, Drilling Down Into Resources

If you click on a DEX file, such as `classes.dex`, a bottom panel displays details of the classes inside the file:

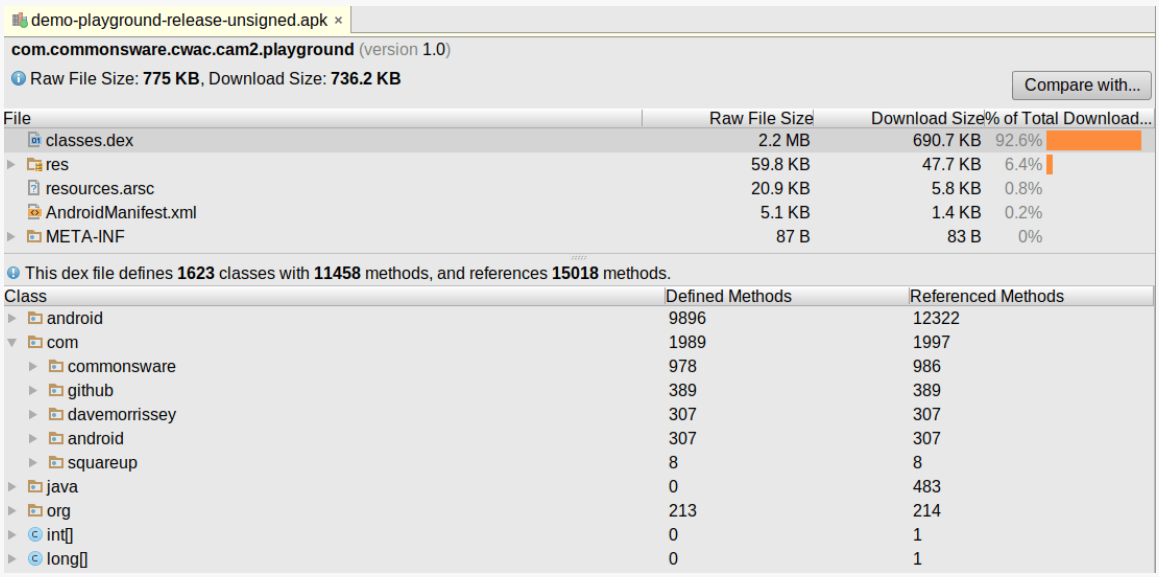


Figure 1061: APK Analyzer, Drilling Down Into DEX

The "Defined Methods" column indicates how many methods are defined by the classes in the packages, where those classes are in the DEX file. "Referenced Methods" refers to how many methods that classes in the DEX file reference from the indicated packages, and that ties into what's known as the "64K DEX method reference limit", which we will get into a bit later in this chapter.

If you have multiple builds of your APK — such as a debug and a release build, or an old and a new build — you can open one in the APK Analyzer, then click the "Compare with..." button and choose the second APK. This opens up a floating window showing you the size differences between the two APKs:

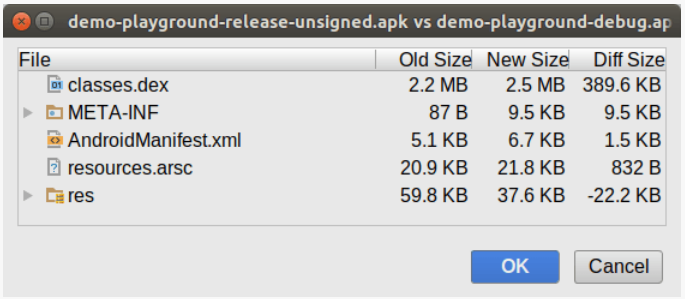


Figure 1062: APK Analyzer, Comparing Two APKs

Java Code, and the 64K Method Limit

In ordinary Java development, there are few limits as to how big your applications can get. You tend to run into physical limitations, such as available system RAM, before you run into any limitations of the programming language or runtime environment.

And, normally, in Android applications, you do not worry about how many classes or methods you have. However, "normally" is not "always", and there is a specific scenario that complex apps need to worry about.

What Is It?

Quoting Andy Fadden, Android platform engineer:

> The issue is not with the Dalvik runtime nor the DEX file format, but with the current set of Dalvik instructions.
>
> You can reference a very large number of methods in a DEX file, but you can only invoke the first 65536, because that's all the room you have in the method invocation instruction.
>
> I'd like to point out that the limitation is on the number of methods referenced, not the number of methods defined. If your DEX file has only a few methods, but together they call 70,000 different externally-defined methods, you're going to exceed the limit.
>
> [An externally-defined method is] a method defined in a separate DEX file. For most apps this would just be framework and core library / uses-library stuff.

Specifically, you will crash at compile time, with an error message akin to:

```
Unable to execute dex: method ID not in [0, 0xffff]: 65536
Conversion to Dalvik format failed: Unable to execute dex: method ID not
in [0, 0xffff]: 65536
```

64K Seems Like a Lot of Typing…

Well, it is, and it isn't.

First, it is not merely your own methods. You can reach the 64K method limit without implementing 64K methods in your application yourself. You can:

- Call lots of methods defined by the framework
- Absorb lots of methods from libraries, particularly larger libraries that offer many more features than your app uses

This still tends to mean that simpler apps are unlikely to run into this limit, while more complex apps might.

Where Are The Methods Coming From?

The APK Analyzer can help you determine what methods get referenced in your APK file. Just click on the DEX file and start looking through the package tree, specifically looking for methods that will not be in your DEX file, such as `java.*`, `javax.*`, and non-`support` classes in `android.*`:

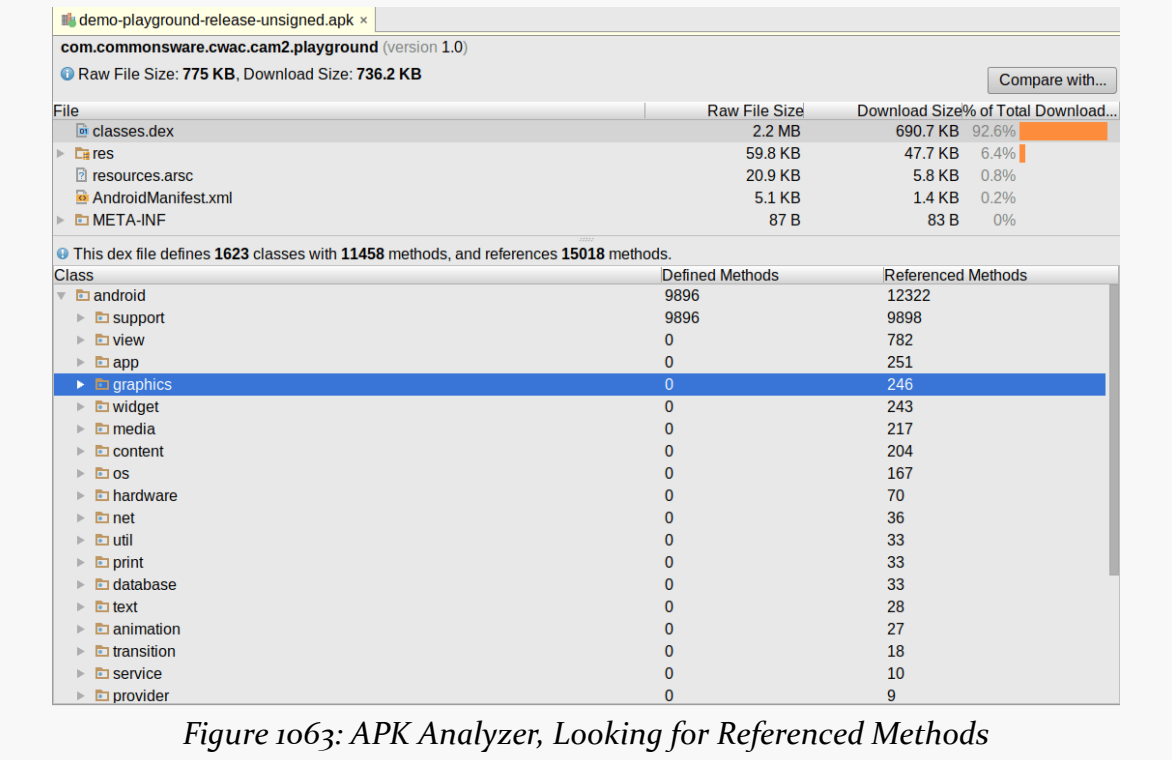


Figure 1063: APK Analyzer, Looking for Referenced Methods

For example, the app being profiled in the APK Analyzer in these screenshots reportedly references four methods on `RippleDrawable`:

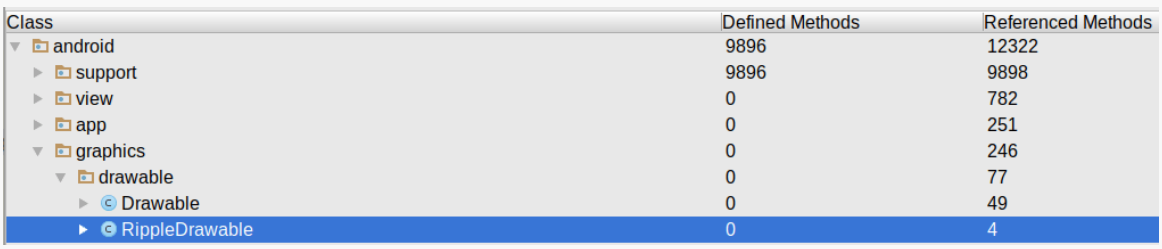


Figure 1064: APK Analyzer, Showing Referenced Methods on RippleDrawable

The developer of this app (who also happened to write this book) does not reference `RippleDrawable` anywhere. Hence, some library that the app is pulling in has code tied to `RippleDrawable`.

Mitigation Tactics

If you are relatively close to the 64K method limit, you may be able to tweak your project to get back under the limit without having to significantly rework your project.

Use Granular Libraries

Some libraries, like Google Play Services, come in two forms: a "kitchen sink" and more granular libraries for individual features. If your need for the library can be met by the granular libraries, use them, and you can remove your dependency on the "kitchen sink".

In the case of Google Play Services, try *not* to depend upon the `com.google.android.gms:play-services` artifact. Instead, try to depend upon one of the more granular artifacts, such as `com.google.android.gms:play-services-maps` for Maps V2. For services, like Google Cloud Messaging, that have no specific granular artifact, depend instead upon `com.google.android.gms:play-services-base` — while still large, this is far smaller than is `com.google.android.gms:play-services`.

Use Better Libraries

One common culprit of hitting the 64K method limit comes from libraries, as their methods count along with yours. Hence, choosing different libraries can perhaps reduce your method count.

One specific case of this comes from the code generated by Google's Protocol Buffers. If you are using Protocol Buffers heavily, your generated classes may each be defining hundreds of unused methods. Switching to an alternative implementation can reduce this significantly. Some such implementations include:

- micro-protobuf
- Square's Wire

Use ProGuard

If your debug builds are failing due to the 64K method limit, try a release build. If that works, the reason is ProGuard and its ability to strip out code that is deemed to be unreachable.

In this case, you can "buy yourself some time" by arranging to build your app in debug mode with ProGuard, but without ProGuard's normal code obfuscation work (e.g., `-dontoptimize -dontobfuscate` switches in the ProGuard configuration). Quoting Eric Lafortune, ProGuard's lead developer:

> If you apply ProGuard with shrinking enabled but optimization and obfuscation disabled (-dontoptimize -dontobfuscate), the code will already be more compact, and you can still use a debugger. The source files, class names, method names, line numbers, etc remain unchanged and any breakpoints in removed unreachable code are irrelevant.

With Gradle-based builds, it should be possible to set this up using `minifyEnabled true` and a custom ProGuard configuration file:

```
android {
    buildTypes {
        debug {
            minifyEnabled true
            proguardFile 'proguard-no-obfuscate.txt'
        }
    }
}
```

(where `proguard-no-obfuscate.txt` contains the `-dontoptimize -dontobfuscate` switches)

Mitigation Strategies

If the aforementioned tactics are insufficient — or if they help somewhat, but you are still near the limit with a lot of development yet to be done — you may need to pursue some more strategic ways of resolving your application size.

Don't Go Overboard

One source of method explosion comes from too much adherence to server-side Java coding styles.

For example, if you find yourself defining hundreds of interfaces and/or abstract classes, with `Factory` classes (and perhaps `FactoryFactory` classes), you are more likely to hit the 64K method limit due to all those separate definitions. Consider whether the flexibility that you believe that you obtain from this coding style is worth the risk.

Smaller Apps, Loosely Connected

It may be that you are simply creating an app that is entirely too complicated for the Android environment. Android's `Intent` system is designed to enable apps to inter-operate, and so you may need to consider splitting your app into pieces, such as:

- A suite of related apps
- A host app and plugin apps that enable additional functionality
- An app and an affiliated Web app, where certain functionality is handled by the Web app in a standard browser

Multidex

The 64K method reference limit refers to the number of methods referenced from a given DEX file. One way to address this, in theory, would be to split the app into multiple DEX files, each getting some of the classes from your app and its dependencies. Each DEX file is likely to be below the 64K method reference limit.

Android has native multidex support starting with Android 5.0. There is a quasi-backport of this for older devices, but it opens the door to security issues and ideally is avoided.

To support native Android 5.0+ multidex, you need to add `multiDexEnabled true` to the `defaultConfig` closure in your module's `build.gradle` file, alongside where you declare your `minSdkVersion`, `targetSdkVersion`, and so on.

Native Code

Native code, implemented as NDK-compiled libraries, represent another source of app bloat. This will occur regardless of whether the NDK code is yours or if you are using a third-party library that supplies those binaries (e.g., SQLCipher for Android).

Native code is not intrinsically large. However, in some cases, native code is a port from some other environment (or environments) and may contain a lot of stuff that your app does not need. Worse, ProGuard will not strip out unused native code, as its algorithms only work with Java-style bytecode. Hence, it is not out of the question for apps to devote several MB just to the Linux `.so` files that make up the NDK-compiled libraries.

Fortunately, there are some workarounds.

Mitigation via Per-CPU APKs

Some distribution channels, like the Play Store, support publishing multiple versions of an APK, with different versions for different CPU architectures. Hence, you could have one APK with x86 binaries and one APK with ARM binaries, as opposed to having one "fat binary" with both.

The Android Gradle Plugin offers APK splits as a way of implementing this, where you get different APKs based on CPU architecture.

Mitigation via libhoudini

As is noted in the chapter on the NDK, `libhoudini` is proprietary Intel code that allows ARM-compiled NDK binaries to run on x86 CPUs, using the same sort of opcode translation that is used by the Android emulator. Many, though not all, x86-powered Android devices have `libhoudini`. Those that do could run your app

even if you only ship ARM NDK binaries and not x86 ones. This gives you the same sort of space savings as you would get by publishing separate ARM vs. x86 APKs (per the previous section), without having to manage multiple APKs yourself. The cost is speed, as the translation layer adds significant overhead, much as you see with the Android emulator running ARM emulator images instead of x86 ones.

Mitigation via Ignoring Non-ARM

Of course, what a lot of developers do is simply only worry about ARM.

While Google does not publish percentages of CPU architectures the way they do Android OS versions, it is safe to say that, as of early 2014, ~1% of Android devices are powered by non-ARM CPUs. That percentage may climb, particularly as Intel pushes more x86 chipsets. But the *vast* majority of Android devices are powered by ARM. So, even if some of those x86 environments lack `libhoudini` (e.g., the manufacturer did not license `libhoudini` from Intel), they are so few in number that developers are prone to ignore x86.

Ironically, what drives x86 for developers is the development environment itself, not the production environment. The x86 emulator is nicely responsive, compared to a similarly-configured ARM emulator image. Many developers avoid the ARM emulator entirely, with it being too slow. Hence, developers may be interested in having x86 binaries in the APK to allow the app to run on the x86 emulator (which lacks `libhoudini`). In this case, it may be worthwhile to have a dedicated release build process that strips out the x86 binaries, if the space that those binaries take up is more than you can afford.

Images

Bitmap images are notorious for taking up lots of heap space. However, they can also swell the size of your APK. While the bitmap PNG or JPEG files will be compressed on disk, if you have enough of them, they can still consume many MB of space in the APK, particularly since the APK cannot compress them further.

Mitigation via Resource Aliases

You may have multiple copies of the same image.

The example cited in the Android documentation is where you want to have locale-specific drawable images. For example, perhaps you want to show a flag, and you use

language resource sets to try to map the right flag to the right language. However, some flags are going to be used in multiple languages, such as the Canadian flag being needed for `en-rCA` and `fr-rCA`. By default, you would place your flag icon in each of those resource sets, duplicating your results. This gets worse if you have a few versions of the same flag icon for different densities.

However, you can elect to use a resource aliases to handle this differently.

Suppose that your code refers to a `flag` drawable resource (e.g., `@drawable/flag` or `R.drawable.flag`). For many languages, you would have a unique flag in the appropriate resource set. For cases where the same flag is used in multiple situations:

1. Put the flag in a resource set that is not tied to locale (e.g., `res/drawable-hdpi/` instead of `res/drawable-en-rCA-hdpi/`), for as many densities as you choose, but under a different name (e.g., `flag_canada.png` instead of `flag.png`)
2. Create a small XML file, `flag.xml`, in each of the locale-specific directories (e.g., `res/drawable-en-rCA/` and `res/drawable-fr-rCA/`), pointing to your `flag_canada` drawable:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/flag_canada" />
```

When you reference `R.drawable.flag` on an `en-rCA` or `fr-rCA` device, Android will read in the XML resource, then turn around and retrieve the `flag_canada` drawable, and use that. Since the two XML files are likely to be smaller than the sum total of the duplicate copies, you save disk space.

Mitigation via pngquant

In practice, the above technique is just not that commonly used, because it addresses a fairly narrow scenario. A more general-purpose solution is to try to tweak the images to be visually nearly identical, yet take up less disk space.

There are a variety of tools for this, mostly aimed at Web development, where smaller image file sizes means faster-loading Web pages.

One such tool is pngquant. Given a PNG file as input, it generates a smaller PNG file as output, one with an optimized color palette, using mathematical techniques to choose colors that will maintain as much of the original look as possible. Many of the images in this book were optimized using `pngquant`, at a substantial savings in disk size, without materially sacrificing image quality.

APK Expansion Files

The ultimate solution to disk space concerns, for distribution through the Play Store, is to get stuff out of your app entirely and distribute that stuff by other means. The Play Store offers APK expansion files with this in mind. You can publish one or two expansion files, each containing up to 2GB of files. While these will not be treated as resources or assets, you do have access to the file contents at runtime. Game developers will use these for sound effects, additional artwork, and so on. The biggest limitation is that these files may not be supported by all distribution channels.

Crash Reporting Using ACRA

When you wrote your app, you intended for it to work.

Alas, the road to a very warm place is paved with good intentions.

Hence, it is fairly likely that your app will crash in the hands of your users. In order to be able to fix the underlying problems, you need to learn about the crashes and the state of the app at the time of the crash.

There are any number of solutions to this problem. This chapter will outline a few of them and focus on one open source solution: Application Crash Reports for Android, better known as ACRA.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Having read the chapter on notifications is also a good idea, though not absolutely essential.

What Happens When Things Go "Boom"?

In development, when your app crashes, you get a little dialog box indicating that the app crashed, and you get your Java stack trace in Logcat.

In production, little of that does you any good. In particular, you have no way of seeing Logcat from end user devices. Instead, you need to have some means of capturing that stack trace, along with perhaps additional data, and collect it somewhere.

App distribution channels may offer this as part of their feature set. The Play Store, in particular, offers its own crash reporting, where crashes "in the field" get reported to you by means of your Developer Console on the Web. However:

- You might not be distributing through the Play Store at all, let alone exclusively, and so the Play Store reporting does not help you for all your users
- The Play Store's approach makes reporting the crash optional, as the user can elect to not send a report, meaning that you don't find out about every crash
- You have no control over what data is and is not collected, both for ensuring that you have enough information to have a shot at fixing the bug and for minimizing extraneous data that might have privacy implications
- Google gets a copy of the crash data, which you may or may not find to be appropriate

Various other services, from Crashlytics to Crittercism, offer their own crash reporting as part of a larger suite of features. However, once again, you may not have control over what data is collected, and you certainly have no control over who all gets the data.

For the privacy-minded app developer, you want something along these lines, but where you can control to a fine degree of detail what gets collected and where the data is sent solely to you, not to some third party.

And that's where ACRA comes in.

Introducing ACRA

ACRA has been around since 2010, originally on Google Code, and now on GitHub. It comes in the form of a library that you add to your app, with code that will get control when an unhandled exception occurs inside your app. There, ACRA *carefully* will collect information about the crash (e.g., the stack trace) and the environment (e.g., what version of Android the app was running on). ACRA can then deliver that information to you by any number of means, plus optionally provide feedback to the user about the crash itself.

Since you control what ACRA collects and you control where ACRA sends the data, you can minimize how much information gets into the hands of third parties. The cost is in convenience, as either you have to:

- Fuss with managing your own server for receiving the crashes, or
- Use a third-party service for that server, reducing some of the privacy, or
- Use options that are clunky for everyone involved, such as the user sending emails containing crash reports

Where ACRA Reports Crashes

In the beginning, ACRA logged crashes to a Google Docs spreadsheet. Eventually, Google grumbled about this, and so that option is now deprecated.

That limitation notwithstanding, ACRA supports a range of possible ways for crash reports to get from the user's device to your eyes, so that you can try to fix whatever problems ail your app.

An Existing Crash Logging Service

Some crash logging services allow you to use ACRA in your code, rather than rely upon some proprietary library. You simply configure ACRA to send the data to their servers, which then notify you about crashes and give you dashboards and such to visualize how much your app is crashing.

HockeyApp and Splunk Mint are two such services.

The advantage here is convenience coupled with control over the client side. However, you are still sharing crash details with third parties, potentially raising privacy or security issues.

Acralyzer

The official ACRA reporting server is Acralyzer. This, along with its `acra-storage` companion, are CouchApps, powered by Apache CouchDB. You upload the Acralyzer and `acra-storage` CouchApps into your own CouchDB instance, then configure ACRA in your app to talk to those apps.

Acralyzer and `acra-storage` are open source, as is CouchDB. You can either host a CouchDB instance on your own server or use various CouchDB hosting providers.

This solution offers the best blend of analysis features and user privacy and security. However, it does require you to learn enough about CouchDB to be able to set up and maintain an instance.

Email

The easiest solution to set up is the most awkward for everything else: have the user send you an email. In this model, ACRA prepares a report, then uses `ACTION_SENDTO` to lead the user to an email app to send the report to an email address that you configure in your app. The user can then just send the prepared email from their email client (e.g., Gmail), and the report shows up in the inbox for this email address.

You do not need to set up some sort of server, let alone maintain it. Your app does not even need the `INTERNET` permission.

However:

- The user might not send the email, choosing instead to abandon the mail client
- The user might not use their device for email, and therefore have no good means of getting you the report
- While you get the raw crash data, you do not get any of the nifty charts and such that you can get from a full-fledged crash reporting server

A Host for Testing

The protocol used by ACRA to communicate with a Web server is blissfully simple. Handling ACRA crash reports yourself does not require that much server-side code, in case you wanted to integrate this capability into the rest of your REST-style Web services.

For example, this trivial Ruby script implements an ACRA-compatible endpoint:

```
require 'fileutils'
require 'sinatra'
require 'json'

LOG_ROOT='/tmp/ACRAfier'

put '/reports/:id' do
  acra=JSON.parse(request.body.read)
  FileUtils.mkdir_p(LOG_ROOT) if !File.exist?(LOG_ROOT)

  f=File.join(LOG_ROOT, params[:id]+'.json')
  File.open(f, 'w') {|io| io.write(JSON.pretty_generate(acra))}
end
```

(from ACRA/Simple/stub_server.rb)

As we will see later in this chapter, you can configure ACRA to use a simple HTTP `PUT` request to submit a crash report to the server. This Ruby script implements a small REST-style Web service using Sinatra, where crash reports are pushed to a `/reports/.../` URL, where `...` is an ACRA-generated unique ID for the report. This script just logs the JSON that we get from ACRA to a file in a designated directory. With a few more lines of code, you could have it generate a human-readable report and email it to you, along with the JSON as an email attachment. Or, you could do whatever you want.

This Ruby script can be found as `stub_server.rb` in the book's GitHub repo If you have Ruby installed, just install the `sinatra` and `json` gems, then run `ruby stub_server.rb` to fire up the server.

In practice, you would need a bit more smarts on a publicly-visible Web service, to help prevent people from maliciously flooding your crash reporting server with bogus data. However, the minimal requirements for ACRA are very straightforward and could be implemented in any reasonable server-side Web framework.

ACRA Integration Basics

Given that you have identified how you want to receive the crash reports, the next step is to add ACRA to your project and configure it to send crash reports to your chosen location.

The `ACRA/Simple` sample project demonstrates a fairly simple ACRA integration.

Adding the Dependency

ACRA is distributed through standard Maven-style artifact repositories and should be automatically picked up when you add the appropriate `implementation` directive to your `dependencies`:

```
dependencies {
    implementation 'ch.acra:acra:4.9.1'
}
```

(from ACRA/Simple/app/build.gradle)

Build Types, Product Flavors, and ACRA

It is very likely that you will want to have different ACRA configurations based upon build types and/or product flavors:

- Have the `debug` build not use ACRA, but have the `jenkins` build by your CI server use ACRA to collect crashes and integrate them into the test results, and have the `release` build use your production ACRA server
- Skip ACRA for your Play Store distribution (because you decide you would rather just use the Play Store's crash reporting), but use ACRA for your `amazon` product flavor (the version of your app that you distribute through the Amazon AppStore for Android)
- And so on

`buildConfigField` is a great way to manage this. Use your `build.gradle` file to establish values for some constants, then use them in the ACRA configuration code in Java later on.

The sample app defines two such fields for `BuildConfig`:

- `ACRA_INSTALL`, a `boolean` that will be `true` if we should use ACRA, `false` otherwise
- `ACRA_URL`, a `String` that will point to the server to which we wish to push the ACRA-collected crash data

The sample app defines the same values for both fields in both build types (`debug` and `release`), simply because you are probably playing around with the sample in a `debug` build:

```
    buildTypes {
        debug {
            buildConfigField "String", "ACRA_URL", '"http://10.0.2.2:4567/reports"'
            buildConfigField "boolean", "ACRA_INSTALL", 'true'
        }

        release {
            buildConfigField "String", "ACRA_URL", '"http://10.0.2.2:4567/reports"'
            buildConfigField "boolean", "ACRA_INSTALL", 'true'
        }
    }
```

(from ACRA/Simple/app/build.gradle)

The URL used for `ACRA_URL` points to 10.0.2.2, the IP address on an Android emulator that refers back to the `localhost` of your developer machine. In particular,

this URL is set up for the server Ruby script mentioned previously in this chapter. If you wish to use a different server, not only will you need to consider changing this URL, but you will need to make some other adjustments to the Java code, in all likelihood, as will be seen in the next couple of sections.

Creating a Custom Application

ACRA needs some one-time initialization, and it is set up to do that by means of a custom `Application` subclass. Most likely, you do not already have one of these, though some libraries will require you to create one, perhaps inheriting from some library-supplied `Application` subclass.

Regardless, you will need a subclass of `Application` in your project, and you will need to have the `android:name` attribute of your `<application>` element in the manifest point to that `Application` subclass:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.button"
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.INTERNET" />

  <uses-sdk
    android:minSdkVersion="21"
    android:targetSdkVersion="21" />

  <application
    android:name=".ACRAApplication"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".ButtonDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
```

```
</manifest>
```

(from ACRA/Simple/app/src/main/AndroidManifest.xml)

Here, `android:name` points to an `ACRAApplication` class that we will examine shortly.

Also note that the manifest has a `<uses-permission>` element, asking for the `INTERNET` permission. Unless you use ACRA's built-in support for sending crash reports via the user's email app, you will need the `INTERNET` permission for getting crash reports to some server.

Implementing the Application

The `Application` subclass that you create needs two items to configure ACRA:

1. A `@ReportsCrashes` annotation, providing the actual ACRA configuration itself
2. A call to `ACRA.init()` from `onCreate()`, to initialize the ACRA crash-detection subsystem and have it use the annotation to configure what to do when crashes occur

```
package com.commonsware.android.button;

import android.app.Application;
import org.acra.ACRA;
import org.acra.annotation.ReportsCrashes;

@ReportsCrashes(
  formUri=BuildConfig.ACRA_URL,
  httpMethod=org.acra.sender.HttpSender.Method.PUT,
  reportType=org.acra.sender.HttpSender.Type.JSON
)
public class ACRAApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    if (BuildConfig.ACRA_INSTALL) {
      ACRA.init(this);
    }
  }
}
```

(from ACRA/Simple/app/src/main/java/com/commonsware/android/button/ACRAApplication.java)

`@ReportsCrashes` has many knobs to turn and switches to flip as part of configuring how ACRA should behave. We will look at a number of them in this chapter. This simple sample configures ACRA to:

- format the crash data as JSON (`reportType=org.acra.sender.HttpSender.Type.JSON`)
- send it to the server indicated by the `BuildConfig.ACRA_URL` value we configured in Gradle (`formUri=BuildConfig.ACRA_URL`)
- use an HTTP `PUT` operation to hand that JSON over to that server (`httpMethod=org.acra.sender.HttpSender.Method.PUT`)

These values work great with the Ruby script profiled earlier in this chapter. If you use some other server, you may need to change this configuration to match what that server wants.

Note that the `ACRA.init()` call is inside a check of the `BuildConfig.ACRA_INSTALL` `boolean` that we set up in the Gradle build files. If a particular build type or product flavor sets `ACRA_INSTALL` to `false`, ACRA will not be enabled. For simpler projects, rather than defining your own `ACRA_INSTALL`-style flag, you could just use `!BuildConfig.DEBUG`, to only configure ACRA on `release` builds. While there is nothing stopping you from using ACRA in development, you may find that it interferes somewhat with how you are used to debugging your crashes.

The approach shown here, using `onCreate()`, is fine for most apps. If your app has a `ContentProvider`, and it does significant work in its own `onCreate()` method, though, you will have a problem: `onCreate()` of `Application` is called *after* `onCreate()` of the provider(s). This means that any crashes in your providers' `onCreate()` will not be caught or reported by ACRA. You can try overriding `attachBaseContext()`, instead of `onCreate()`, in your `Application` subclass and initializing ACRA there, as `attachBaseContext()` appears to be called before `onCreate()` of any providers.

Also, if you have an existing `Application` subclass, you need to consider how ACRA's error-reporting process will impact your existing logic.

Reporting Crashes

Good news! You're done!

ACRA does not require you to litter your code with magic `try/catch` blocks to catch and report exceptions. After all, some Android exceptions – even those triggered from bugs in your code — are raised by Android framework code and your code appears nowhere in the stack trace.

Instead, ACRA takes advantage of `Thread` and its `setDefaultUncaughtExceptionHandler()` method, to get control when any unhandled exception occurs. All those crashes that normally would shut down a component or the whole app now go to ACRA and can be reported to your designated server.

Occasionally, you may wish to add some crashes that you *are* handling yourself to ACRA. For example, there may be some edge or corner cases that you are explicitly handling but are uncertain if they ever would happen. You could arrange to pass the `Exception` over to ACRA, which it will treat the same as any other crash that it intercepts.

To do this, call `getErrorReporter()` on the `ACRA` class, and call either `handleException()` or `handleSilentException()` on the error reporter. The difference is that `handleSilentException()` always reports the error silently, while `handleException()` will process this exception like any other, possibly alerting the user to the crash, as will be seen in the next section.

What the User Sees

The `Simple` sample app has ACRA configured, but this does us little good if we do not crash. So, the UI for the activity has a `Button`, and tapping that button will trigger a `RuntimeException`:

```
package com.commonsware.android.button;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;

public class ButtonDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
```

```
  public void earthShatteringKaboom(View v) {
    throw new RuntimeException(getString(R.string.msg_kaboom));
  }
}
```

(from ACRA/Simple/app/src/main/java/com/commonsware/android/button/ButtonDemoActivity.java)

...whose message is tied to a string resource.

When you click the `Button`, ACRA will send a crash report to your designated server. What the *user* perceives, though, varies based upon configuration.

Default: "Silent"

If you do not specify otherwise in your ACRA configuration, the default behavior will be "silent". In this case, "silent" means "the user is not told that a report is being sent via ACRA". Instead, the user sees the traditional Android crash dialog, for whatever version of Android the app is running on:

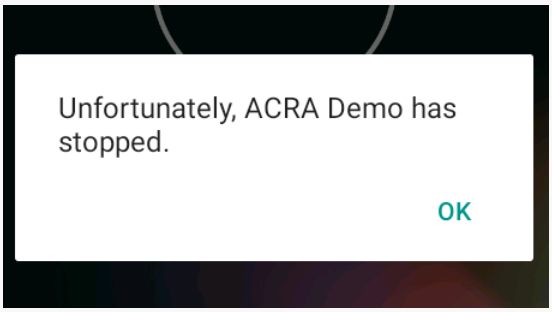


Figure 1065: ACRA-Reported Crash, Silent Mode

However, there are several options that you can use instead of "silent" mode, if you so choose. One — showing a `Toast` — is not an especially good idea, as the user might not be looking at the screen right then and might not see the message.

Dialog

Another option is the "dialog" approach, where the user is shown a dialog-themed activity, indicating what happened and allowing the user to provide some additional information.

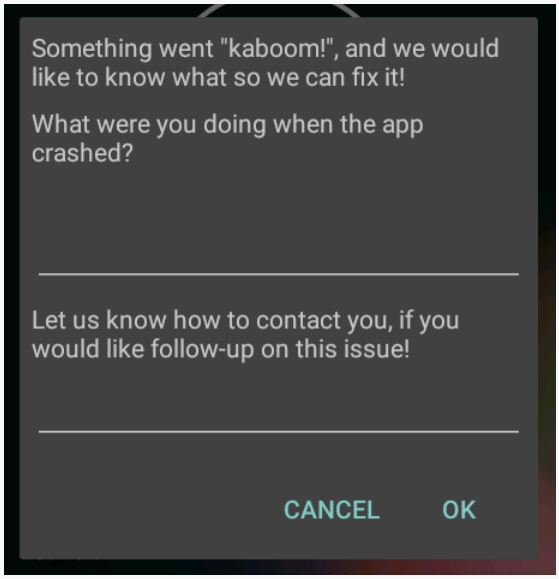


Figure 1066: ACRA-Reported Crash, Dialog Mode

On the plus side, this is more transparent to the user, and the user can provide a bit more detail that might be useful to you. However, the user can also cancel out of the dialog, in which case you do not receive a crash report at all.

To set this up, you need to add a few more options to your ACRA configuration. You can see this in `ACRADialogApplication` in the sample project, which is a clone of `ACRAApplication`, set up for dialog-style reporting:

```
package com.commonsware.android.button;

import android.app.Application;
import org.acra.ACRA;
import org.acra.ReportingInteractionMode;
import org.acra.annotation.ReportsCrashes;

@ReportsCrashes(
  formUri=BuildConfig.ACRA_URL,
  mode = ReportingInteractionMode.DIALOG,
  resToastText = R.string.msg_acra_toast,
  resDialogText = R.string.msg_acra_dialog,
  resDialogCommentPrompt = R.string.msg_acra_comment_prompt,
  resDialogEmailPrompt = R.string.msg_acra_email_prompt,
  resDialogTheme = R.style.AppTheme_Dialog,
  httpMethod=org.acra.sender.HttpSender.Method.PUT,
  reportType=org.acra.sender.HttpSender.Type.JSON
)
```

```
public class ACRADialogApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    if (BuildConfig.ACRA_INSTALL) {
      ACRA.init(this);
    }
  }
}
```

(from ACRA/Simple/app/src/main/java/com/commonsware/android/button/ACRADialogApplication.java)

What turns on dialog mode is the `mode = ReportingInteractionMode.DIALOG` entry in the `@ReportsCrashes` annotation. This requires one additional entry, `resDialogText`, pointing to a string resource that is the message to display towards the top of the dialog.

You have a number of other optional settings to use to further customize the dialog. `ACRADialogApplication` demonstrates:

- `resToastText`, a string resource that will be shown in a `Toast` after the crash occurs and before the dialog appears. It takes ACRA a few seconds to collect the data for the crash report, and ACRA does not display the dialog until that data is collected. The `Toast` lets the user know that something is going on during this window of time.
- `resDialogCommentPrompt`, a string resource which, if included in `@ReportsCrashes`, enables a large `EditText` widget where the user can type in some comments about what they were doing at the time of the crash. The string resource serves as a label for this `EditText`.
- `resDialogEmailPrompt`, a string resource which, if included in `@ReportsCrashes`, enables an `EditText` widget where the user can type in an email address or other means of contacting the user. This value is saved in an ACRA-specific `SharedPrefererences` value, and so it may already be filled in for the user, if the user had supplied a value previously. This, along with the comments, is included in the crash report for your use. The string resource serves as a label for this `EditText`.
- `resDialogTheme`, pointing to the theme that you would like to use for the dialog, such as `R.style.AppTheme_Dialog` theme to reference what, in the manifest, you would call `@style/AppTheme.Dialog`

This sample app only runs on API Level 21+ (as it depends upon `Theme.Material` for the main UI), so we only need to provide one theme definition, here called `AppTheme.Dialog`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>

  <style
    name="AppTheme.Dialog"
    parent="@android:style/Theme.DeviceDefault.Dialog"/>
</resources>
```

(from ACRA/Simple/app/src/main/res/values/styles.xml)

Here, we follow ACRA's advice and have `AppTheme.Dialog` inherit from `Theme.DeviceDefault.Dialog`. `DeviceDefault` is a theme based on the core theme for the Android OS version (`Material` for Android 5.0+), but one that can be tailored by device manufacturers and custom ROM developers. By extending `Theme.DeviceDefault.Dialog`, we are saying that we want our dialog to be styled like other system dialogs.

`Theme.DeviceDefault.Dialog` should be a fine base theme for API Level 11+. If you are supporting older Android devices than that, for those older API levels, use `Theme.Dialog` instead.

In the annotation parameters, you can also configure an icon for the dialog (`resDialogIcon`), a title to go across the top of the dialog (`resDialogText`), and the text for a `Toast` to be shown when the user taps OK (`resDialogOkToast`).

Of course, your `android:name` attribute of your `<application>` element in the manifest will need to point to this `Application` subclass. If you wish to try the dialog in the sample app, you will need to modify the sample app's manifest to point to `ACRADialogApplication` instead of `ACRAApplication`.

Notification

While the dialog mode is great, it is unsuitable for crashes that may occur in the background. You do not want to pop a dialog box up unexpectedly, as users may not appreciate the interruption.

The default "silent" mode, for crashes originating in the background, will not show a dialog. This is far more suitable for background work, but it does not let the user know that a crash occurred.

The `Notification` mode serves as middle ground. When a crash occurs in the background, ACRA raises a `Notification`. Tapping on that `Notification`, in turn, brings up the same dialog that the dialog mode uses.

To use this, switch your `mode` to `ReportingInteractionMode.NOTIFICATION` in the `@ReportsCrashes` annotation. Then, in addition to all the dialog configuration, add three more string resource references:

- `resNotifTickerText`, shown as the "ticker text" of the `Notification` on Android 4.4 and below
- `resNotifTitle`, shown as the title of the `Notification` in its tile in the notification tray
- `resNotifText`, shown as the text of the `Notification` in its tile in the notification tray

Optionally, you can also set `resNotifIcon` to a particular drawable resource to use for the icon for the `Notification`.

The sample app has an `ACRANotificationApplication` that demonstrates this:

```
package com.commonsware.android.button;

import android.app.Application;
import org.acra.ACRA;
import org.acra.ReportingInteractionMode;
import org.acra.annotation.ReportsCrashes;

@ReportsCrashes(
  formUri=BuildConfig.ACRA_URL,
  mode = ReportingInteractionMode.NOTIFICATION,
  resToastText = R.string.msg_acra_toast,
  resDialogText = R.string.msg_acra_dialog,
  resDialogCommentPrompt = R.string.msg_acra_comment_prompt,
```

```
  resDialogEmailPrompt = R.string.msg_acra_email_prompt,
  resNotifTickerText = R.string.msg_acra_notify_ticker,
  resNotifTitle = R.string.msg_acra_notify_title,
  resNotifText = R.string.msg_acra_notify_text,
  httpMethod=org.acra.sender.HttpSender.Method.PUT,
  reportType=org.acra.sender.HttpSender.Type.JSON
)
public class ACRANotificationApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    if (BuildConfig.ACRA_INSTALL) {
      ACRA.init(this);
    }
  }
}
```

(from ACRA/Simple/app/src/main/java/com/commonsware/android/button/ACRANotificationApplication.java)

If you switch the `android:name` of the `<application>` manifest element over to point to `ACRANotificationApplication`, crashing the app will bring up the `Notification`:

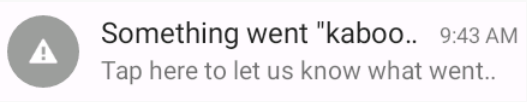


Figure 1067: ACRA-Reported Crash, Notification Mode

(pro tip: use short strings)

Limitations

The big limitation is that you get exactly one reporting mode for your app, for automatically-collected crashes. This means that your choice of reporting mode will be dictated by whether or not you are doing work in the background, while you do not have a UI in the foreground (e.g., a `Service`):

- If you are not doing background work, use the dialog or silent modes
- If you are doing background work, use the notification or silent modes

What You See

The sample app asks ACRA to send the crash data over in a JSON structure. That JSON contains all sorts of information by default, including `USER_COMMENT` and `USER_EMAIL` properties if you chose the dialog or notification modes.

Here is what we get from a crash of the sample app, using the dialog notification mode:


```
{
  "REPORT_ID": "7583e142-024d-4596-ba83-1a2cb6ae266d",
  "APP_VERSION_CODE": 1,
  "APP_VERSION_NAME": "1.0",
  "PACKAGE_NAME": "com.commonsware.android.button",
  "FILE_PATH": "/data/user/0/com.commonsware.android.button/files",
  "PHONE_MODEL": "Android SDK built for x86",
  "ANDROID_VERSION": "6.0",
  "BUILD": {
    "BOARD": "unknown",
    "BOOTLOADER": "unknown",
    "BRAND": "generic_x86",
    "CPU_ABI": "x86",
    "CPU_ABI2": "",
    "DEVICE": "generic_x86",
    "DISPLAY": "sdk_phone_x86-eng 6.0 MASTER 2401146 test-keys",
    "FINGERPRINT": "generic_x86/sdk_phone_x86/generic_x86:6.0/MASTER/...",
    "HARDWARE": "goldfish",
    "HOST": "kpfj8.cbf.corp.google.com",
    "ID": "MASTER",
    "IS_DEBUGGABLE": true,
    "MANUFACTURER": "unknown",
    "MODEL": "Android SDK built for x86",
    "PRODUCT": "sdk_phone_x86",
    "RADIO": "unknown",
    "SERIAL": "unknown",
    "SUPPORTED_32_BIT_ABIS": "[x86]",
    "SUPPORTED_64_BIT_ABIS": "[]",
    "SUPPORTED_ABIS": "[x86]",
    "TAGS": "test-keys",
    "TIME": 1446737966000,
    "TYPE": "eng",
    "UNKNOWN": "unknown",
    "USER": "android-build",
    "VERSION": {
      "ACTIVE_CODENAMES": "[]",
      "BASE_OS": "",
```

```
    "CODENAME": "REL",
    "INCREMENTAL": 2401146,
    "PREVIEW_SDK_INT": 0,
    "RELEASE": "6.0",
    "RESOURCES_SDK_INT": 23,
    "SDK": 23,
    "SDK_INT": 23,
    "SECURITY_PATCH": "2015-10-01"
  }
},
"BRAND": "generic_x86",
"PRODUCT": "sdk_phone_x86",
"TOTAL_MEM_SIZE": 567640064,
"AVAILABLE_MEM_SIZE": 442961920,
"BUILD_CONFIG": {
  "ACRA_INSTALL": true,
  "ACRA_URL": "http://10.0.2.2:4567/reports",
  "APPLICATION_ID": "com.commonsware.android.button",
  "BUILD_TYPE": "debug",
  "DEBUG": true,
  "FLAVOR": "",
  "VERSION_CODE": 1,
  "VERSION_NAME": ""
},
"CUSTOM_DATA": {
},
"STACK_TRACE": "java.lang.IllegalStateException: Could not execute ...",
"INITIAL_CONFIGURATION": {
  "compatScreenHeightDp": 509,
  "compatScreenWidthDp": 320,
  "compatSmallestScreenWidthDp": 320,
  "densityDpi": 240,
  "fontScale": "1.0",
  "hardKeyboardHidden": "HARDKEYBOARDHIDDEN_NO",
  "keyboard": "KEYBOARD_QWERTY",
  "keyboardHidden": "KEYBOARDHIDDEN_NO",
  "locale": "en_US",
  "mcc": 310,
  "mnc": 260,
  "navigation": "NAVIGATION_NONAV",
  "navigationHidden": "NAVIGATIONHIDDEN_YES",
  "orientation": "ORIENTATION_PORTRAIT",
  "screenHeightDp": 509,
  "screenLayout": "SCREENLAYOUT_SIZE_NORMAL+SCREENLAYOUT_LONG_YES+...",
  "screenWidthDp": 320,
  "seq": 5,
  "smallestScreenWidthDp": 320,
  "touchscreen": "TOUCHSCREEN_FINGER",
```

```
    "uiMode": "UI_MODE_TYPE_NORMAL+UI_MODE_NIGHT_NO",
    "userSetLocale": false
  },
  "CRASH_CONFIGURATION": {
    "compatScreenHeightDp": 509,
    "compatScreenWidthDp": 320,
    "compatSmallestScreenWidthDp": 320,
    "densityDpi": 240,
    "fontScale": "1.0",
    "hardKeyboardHidden": "HARDKEYBOARDHIDDEN_NO",
    "keyboard": "KEYBOARD_QWERTY",
    "keyboardHidden": "KEYBOARDHIDDEN_NO",
    "locale": "en_US",
    "mcc": 310,
    "mnc": 260,
    "navigation": "NAVIGATION_NONAV",
    "navigationHidden": "NAVIGATIONHIDDEN_YES",
    "orientation": "ORIENTATION_PORTRAIT",
    "screenHeightDp": 509,
    "screenLayout": "SCREENLAYOUT_SIZE_NORMAL+SCREENLAYOUT_LONG_YES+...",
    "screenWidthDp": 320,
    "seq": 5,
    "smallestScreenWidthDp": 320,
    "touchscreen": "TOUCHSCREEN_FINGER",
    "uiMode": "UI_MODE_TYPE_NORMAL+UI_MODE_NIGHT_NO",
    "userSetLocale": false
  },
  "DISPLAY": {
    "0": {
      "currentSizeRange": {
        "smallest": "[480,444]",
        "largest": "[800,764]"
      },
      "flags": "FLAG_SUPPORTS_PROTECTED_BUFFERS+FLAG_SECURE",
      "height": 800,
      "name": "Built-in Screen",
      "orientation": 0,
      "pixelFormat": 1,
      "getRealSize": "[480,800]",
      "rectSize": "[0,0,480,800]",
      "refreshRate": 260.416,
      "rotation": "ROTATION_0",
      "getSize": "[480,800]",
      "width": 480,
      "isValid": true
    }
  },
  "USER_COMMENT": "Something",
```

```
"USER_APP_START_DATE": "2015-11-29T09:14:49.000-05:00",
"USER_CRASH_DATE": "2015-11-29T09:14:58.000-05:00",
"DUMPSYS_MEMINFO": "Permission Denial: can't dump meminfo from from ...",
"LOGCAT": "11-29 09:01:46.322 D/ACRA    ( 2076): Looking for error ...",
"INSTALLATION_ID": "44fba689-c636-493c-b95e-07b81806b637",
"USER_EMAIL": "foo@bar.com",
"DEVICE_FEATURES": {
  "android.hardware.sensor.accelerometer": true,
  "android.hardware.faketouch": true,
  "android.software.backup": true,
  "android.hardware.touchscreen": true,
  "android.hardware.touchscreen.multitouch": true,
  "android.software.print": true,
  "android.hardware.ethernet": true,
  "android.software.voice_recognizers": true,
  "android.hardware.camera.autofocus": true,
  "android.hardware.audio.output": true,
  "android.hardware.screen.portrait": true,
  "android.software.home_screen": true,
  "android.hardware.microphone": true,
  "android.hardware.sensor.compass": true,
  "android.hardware.touchscreen.multitouch.jazzhand": true,
  "android.software.app_widgets": true,
  "android.software.input_methods": true,
  "android.software.device_admin": true,
  "android.hardware.camera": true,
  "android.hardware.screen.landscape": true,
  "android.software.managed_users": true,
  "android.software.webview": true,
  "android.hardware.camera.any": true,
  "android.software.connectionservice": true,
  "android.hardware.touchscreen.multitouch.distinct": true,
  "android.hardware.location.network": true,
  "android.software.live_wallpaper": true,
  "android.software.midi": true,
  "android.hardware.location": true,
  "glEsVersion": "0.0"
},
"ENVIRONMENT": {
  "getDataDirectory": "/data",
  "getDownloadCacheDirectory": "/cache",
  "getExternalStorageDirectory": "/storage/1719-3917",
  "getExternalStorageState": "mounted",
  "getLegacyExternalStorageDirectory": "/sdcard",
  "getLegacyExternalStorageObbDirectory": "/sdcard/Android/obb",
  "getOemDirectory": "/oem",
  "getRootDirectory": "/system",
  "getSecureDataDirectory": "/data",
```

```
    "getStorageDirectory": "/storage",
    "getSystemSecureDirectory": "/data/system",
    "getVendorDirectory": "/vendor",
    "isEncryptedFilesystemEnabled": false,
    "isExternalStorageEmulated": false,
    "isExternalStorageRemovable": true
  },
  "SETTINGS_SYSTEM": {
    "ACCELEROMETER_ROTATION": 1,
    "ALARM_ALERT": "content://media/internal/audio/media/9",
    "DTMF_TONE_TYPE_WHEN_DIALING": 0,
    "DTMF_TONE_WHEN_DIALING": 1,
    "HAPTIC_FEEDBACK_ENABLED": 1,
    "HEARING_AID": 0,
    "LOCKSCREEN_SOUNDS_ENABLED": 1,
    "MODE_RINGER_STREAMS_AFFECTED": 422,
    "MUTE_STREAMS_AFFECTED": 46,
    "NOTIFICATION_LIGHT_PULSE": 1,
    "NOTIFICATION_SOUND": "content://media/internal/audio/media/70",
    "POINTER_SPEED": 0,
    "RINGTONE": "content://media/internal/audio/media/105",
    "SCREEN_BRIGHTNESS": 102,
    "SCREEN_BRIGHTNESS_MODE": 0,
    "SCREEN_OFF_TIMEOUT": 60000,
    "SOUND_EFFECTS_ENABLED": 1,
    "TTY_MODE": 0,
    "VIBRATE_WHEN_RINGING": 0,
    "VOLUME_ALARM": 6,
    "VOLUME_BLUETOOTH_SCO": 7,
    "VOLUME_MUSIC": 11,
    "VOLUME_NOTIFICATION": 5,
    "VOLUME_RING": 5,
    "VOLUME_SYSTEM": 7,
    "VOLUME_VOICE": 4
  },
  "SETTINGS_SECURE": {
    "ACCESSIBILITY_DISPLAY_MAGNIFICATION_AUTO_UPDATE": 1,
    "ACCESSIBILITY_DISPLAY_MAGNIFICATION_ENABLED": 0,
    "ACCESSIBILITY_DISPLAY_MAGNIFICATION_SCALE": "2.0",
    "ACCESSIBILITY_SCREEN_READER_URL": "https://ssl.gstatic.com/...",
    "ACCESSIBILITY_SCRIPT_INJECTION": 0,
    "ACCESSIBILITY_SPEAK_PASSWORD": 0,
    "ACCESSIBILITY_WEB_CONTENT_KEY_BINDINGS": "0x13=0x01000100; ...",
    "ANDROID_ID": "23f541e6fcb720b",
    "BACKUP_ENABLED": 1,
    "BACKUP_TRANSPORT": "android/com.android.internal.backup.LocalTransport",
    "DEFAULT_INPUT_METHOD": "com.android.inputmethod.latin/.LatinIME",
    "DOUBLE_TAP_TO_WAKE": 1,
```

```
    "ENABLED_INPUT_METHODS": "com.android.inputmethod.latin/.LatinIME",
    "IMMERSIVE_MODE_CONFIRMATIONS": "",
    "INPUT_METHODS_SUBTYPE_HISTORY": "",
    "INSTALL_NON_MARKET_APPS": 1,
    "LOCK_SCREEN_ALLOW_PRIVATE_NOTIFICATIONS": 1,
    "LOCK_SCREEN_OWNER_INFO_ENABLED": 0,
    "LOCK_SCREEN_SHOW_NOTIFICATIONS": 1,
    "LONG_PRESS_TIMEOUT": 500,
    "MOUNT_PLAY_NOTIFICATION_SND": 1,
    "MOUNT_UMS_AUTOSTART": 0,
    "MOUNT_UMS_NOTIFY_ENABLED": 1,
    "MOUNT_UMS_PROMPT": 1,
    "SCREENSAVER_ACTIVATE_ON_DOCK": 1,
    "SCREENSAVER_ACTIVATE_ON_SLEEP": 0,
    "SCREENSAVER_COMPONENTS": "com.google.android.deskclock/...",
    "SCREENSAVER_DEFAULT_COMPONENT": "com.google.android.deskclock/...",
    "SCREENSAVER_ENABLED": 1,
    "SELECTED_INPUT_METHOD_SUBTYPE": "-1",
    "SELECTED_SPELL_CHECKER": "com.android.inputmethod.latin/...",
    "SELECTED_SPELL_CHECKER_SUBTYPE": 0,
    "SHOW_NOTE_ABOUT_NOTIFICATION_HIDING": 0,
    "SLEEP_TIMEOUT": "-1",
    "TOUCH_EXPLORATION_ENABLED": 0,
    "TRUST_AGENTS_INITIALIZED": 1,
    "USER_SETUP_COMPLETE": 1,
    "WAKE_GESTURE_ENABLED": 1
  },
  "SETTINGS_GLOBAL": {
    "AIRPLANE_MODE_ON": 0,
    "AIRPLANE_MODE_RADIOS": "cell,bluetooth,wifi,nfc,wimax",
    "AIRPLANE_MODE_TOGGLEABLE_RADIOS": "bluetooth,wifi,nfc",
    "ASSISTED_GPS_ENABLED": 1,
    "AUDIO_SAFE_VOLUME_STATE": 1,
    "AUTO_TIME": 1,
    "AUTO_TIME_ZONE": 1,
    "BLUETOOTH_ON": 0,
    "CALL_AUTO_RETRY": 0,
    "CAR_DOCK_SOUND": "/system/media/audio/ui/Dock.ogg",
    "CAR_UNDOCK_SOUND": "/system/media/audio/ui/Undock.ogg",
    "CDMA_CELL_BROADCAST_SMS": 1,
    "CDMA_SUBSCRIPTION_MODE": 1,
    "DATA_ROAMING": 0,
    "DEFAULT_INSTALL_LOCATION": 0,
    "DESK_DOCK_SOUND": "/system/media/audio/ui/Dock.ogg",
    "DESK_UNDOCK_SOUND": "/system/media/audio/ui/Undock.ogg",
    "DEVICE_NAME": "Android SDK built for x86",
    "DEVICE_PROVISIONED": 1,
    "DOCK_AUDIO_MEDIA_ENABLED": 1,
```

```
    "DOCK_SOUNDS_ENABLED": 0,
    "EMERGENCY_TONE": 0,
    "ENHANCED_4G_MODE_ENABLED": 1,
    "GUEST_USER_ENABLED": 1,
    "HEADS_UP_NOTIFICATIONS_ENABLED": 1,
    "LOCK_SOUND": "/system/media/audio/ui/Lock.ogg",
    "LOW_BATTERY_SOUND": "/system/media/audio/ui/LowBattery.ogg",
    "LOW_BATTERY_SOUND_TIMEOUT": 0,
    "MOBILE_DATA": 1,
    "MODE_RINGER": 2,
    "MULTI_SIM_DATA_CALL_SUBSCRIPTION": 1,
    "MULTI_SIM_SMS_SUBSCRIPTION": 1,
    "MULTI_SIM_VOICE_CALL_SUBSCRIPTION": 1,
    "NETSTATS_ENABLED": 1,
    "NETWORK_SCORING_PROVISIONED": 1,
    "PACKAGE_VERIFIER_ENABLE": 1,
    "POWER_SOUNDS_ENABLED": 1,
    "PREFERRED_NETWORK_MODE": 0,
    "SET_INSTALL_LOCATION": 0,
    "STAY_ON_WHILE_PLUGGED_IN": 1,
    "THEATER_MODE_ON": 0,
    "TRUSTED_SOUND": "/system/media/audio/ui/Trusted.ogg",
    "UNLOCK_SOUND": "/system/media/audio/ui/Unlock.ogg",
    "USB_MASS_STORAGE_ENABLED": 1,
    "WIFI_COUNTRY_CODE": "us",
    "WIFI_DISPLAY_ON": 0,
    "WIFI_MAX_DHCP_RETRY_COUNT": 9,
    "WIFI_NETWORKS_AVAILABLE_NOTIFICATION_ON": 1,
    "WIFI_ON": 0,
    "WIFI_SCAN_ALWAYS_AVAILABLE": 0,
    "WIFI_SLEEP_POLICY": 2,
    "WIFI_WATCHDOG_ON": 1,
    "WIRELESS_CHARGING_STARTED_SOUND": "/system/media/audio/ui/..."
  },
  "SHARED_PREFERENCES": {
    "default": {
      "acra": {
        "lastVersionNr": 1
      }
    },
    "": true
  }
}
```


(note: some property values were truncated with . . ., as they were much too long to try to display in a book)

Your server can parse this and use it to take appropriate action.

Note that the Java stack trace (`STACK_TRACE` property) is formatted with embedded Java-style/C-style control characters (`\n` for newlines, `\t` for tabs). Your server can convert that into plain text with appropriate formatting.

Customizing Where Reports Go

The sample app uses one particular approach for sending crash-reports off-device: use an HTTP `PUT` operation, applied to a server configured in `@ReportsCrashes`.

That is not your only option.

HTTP

`httpMethod=org.acra.sender.HttpSender.Method.PUT` in `@ReportsCrashes` is what steers ACRA to use an HTTP `PUT` request to submit the crash report. Without this, by default, it will use an HTTP `POST` request.

However, with `POST`, it treats the URL (in the `formUri` property) a bit differently:

- For a `PUT`, the UUID of the crash report is appended to the URL (`http://localhost:10.0.2.2/reports/f4a411b0-7a5a-0133-dd79-14feb5bc72a7`)
- For a `POST`, the URL is used directly without modification (`http://localhost:10.0.2.2/reports`), where the UUID only appears as the `REPORT_ID` value in the crash report

`reportType=org.acra.sender.HttpSender.Type.JSON` in `@ReportsCrashes` is what tells ACRA to generate a JSON document and submit that as a crash report. If you are using `PUT`, you probably want `JSON`. However, the default (no `reportType`) is a classic Web form encoded string, which would be a more natural choice for `POST` requests, as your server probably already has logic to convert a Web form into more convenient variables in your desired Web app framework and language.

If your server requires HTTP Basic authentication, `formUriBasicAuthLogin` and `formUriBasicAuthPassword` are available as `@ReportsCrashes` properties. Those are of somewhat limited utility, as anyone who can see the whole URL probably can see those HTTP headers without much additional work, and they have to be hard-coded into your app.

Email

Replacing `formUri` and the other HTTP `@ReportsCrashes` properties with `mailTo` (`mailTo=omgomgomg@foo.com`) will cause ACRA to not attempt to deliver the crash report directly. Instead, it will use `ACTION_SENDTO` with a `mailto` `Uri`, pointing at your requested email address, to try to bring up an email client. If the user does not have a configured email client, or if the user chooses not to send the email, you do not get the crash report.

DIY

If none of the stock ACRA delivery options works for you, you are welcome to add your own. You can create an implementation of the `ReportSender` interface, complete with a `send()` method that will be called to actually send the crash report. You can then declare a `reportSenderFactoryClasses` attribute in your `@ReportCrashes` annotation, with an array of `ReportSender` classes:

```
@ReportCrashes{
   reportSenderFactoryClasses = {SpecialReportSender.class}
   // other configuration
}
```

It is up to you to then get the crash report somewhere useful to you, by one means or another.

Adding Additional Data

As demonstrated in the preceding section, ACRA throws a lot of data into the crash report. However, you can add more than that, if you wish, to either better diagnose problems or to provide more individualized assistance.

Adding Stock Data to Emails

With any of the HTTP options, the crash report contains, by default, a lot of information. For email, though, rather than have an attachment with the full report, ACRA only sends along a few bits of data, such as the stack trace.

The `customReportContent` property on `@ReportsCrashes` allows you to tailor this, expanding it to include other report fields. There is a `ReportField` class that defines a series of constants that you use to indicate what should be in the report.

Logcat and Other Logs

ACRA uses some undocumented and unsupported means of collecting Logcat data and including it in the crash report. However, for most Android devices (those running Android 4.1+), this will only contain log lines from your app's process, due to some Android changes, for privacy reasons.

If you have elected to do your own logging elsewhere, you can teach ACRA to incorporate its logs into the crash report:

- Add `ReportField.APPLICATION_LOG` to the list of report fields in the `customReportContent` property on `@ReportsCrashes`
- Add an `applicationLogFile` property on `@ReportsCrashes` to indicate where the log file is
- Optionally add an `applicationLogFileLines` property on `@ReportsCrashes` to indicate how many lines from the log file to include in the crash report (where it defaults to 100)

Note, though, that it is unclear how you express the path to the log file (for `applicationLogFile`), as the actual filesystem path may vary by device and user.

Device Identifier

If your app has the `READ_PHONE_STATE` permission, ACRA will try to include a telephony hardware identifier (e.g., IMEI for GSM phones) in the crash report. However:

- This has privacy implications, and so ACRA has a way to allow the user to control whether this value is included, as will be covered later in this chapter.
- `READ_PHONE_STATE` is a `dangerous` permission, requiring you to request that permission at runtime on Android 6.0+ devices, if your `targetSdkVersion` is 23 or higher. You will not be in position to request this permission at the time of the crash, and so you will need to ask for it at some other point (e.g., on first run of your app).

If your objective is merely to correlate crash reports coming from the same installation of your app, consider storing a UUID in `SharedPreferences` (initialized on first run), as that will be included in your crash report, as is covered in the next section.

Additional SharedPreferences

If you use `PreferenceManager.getDefaultSharedPreferences()`, everything inside of there is included in your ACRA crash report.

If you use other `SharedPreferences` files (e.g., via `getSharedPreferences()`), and you want those `SharedPreferences` included in the crash report, add an `additionalSharedPreferences` property to `@ReportsCrashes`, supplying a list of the preferences filenames:

```
additionalSharedPreferences={"game_stats"}
```

Here, `game_stats` is the `SharedPreferences` filename, passed in as the first parameter to `getSharedPreferences()`.

Your Own Data

ACRA also maintains a process-level `LinkedHashMap` that you can add to, where its contents are included in the crash report. Simply call `ACRA.getErrorReporter().putCustomData()`, supplying the key and value as `String` objects.

Because this is a `LinkedHashMap`, calling `putCustomData()` for some key will replace any past value for that key. The use of `LinkedHashMap` means that the data will be saved (and reported) in alphabetical order. Hence, you are welcome to generate unique keys if you want, perhaps based on `SystemClock.uptimeMillis()`, to use this custom data as an ersatz log.

However, since all of this data is kept in heap space, you will need to be judicious about its use. You are better served using actual file-based logs (whether Logcat or your own) for true logging, reserving this "custom data" for transient state or values that are non-changing.

Note that there is also `removeCustomData()`, which removes a value from the `LinkedHashMap`, given its key. In addition, `getCustomData()` returns the current value given its key, in case you wish to use this `LinkedHashMap` as the master copy of some data, in addition to having that data included in the crash report.

Removing Data

Conversely, you may wish to remove data from the ACRA crash reports. One key reason would be privacy, if there are specific things that you see showing up in the ACRA data that you think users might not like being disclosed. Another reason would be bandwidth, as there is little point in transferring data to be discarded over the Internet, adding load to your servers and perhaps costing your users money on their metered data connections.

Report Fields

As mentioned earlier in this chapter, the `customReportContent` property on `@ReportsCrashes` can be used to add fields to email-based crash reports, which normally only include a small subset of the actual available data.

Conversely, for HTTP-based crash reports — where `customReportContent` defaults to "everything" — `customReportContent` can be used to restrict what is included in the report.

SharedPreference Values

If there are specific `SharedPreferences` values that you would like to be excluded from crash reports, for privacy or security reasons, you can do that via an `excludeMatchingSharedPreferencesKeys` property on `@ReportsCrashes`. For example, if you use `SharedPreferences` to store some limited-life authorization token from a server, it is probably best to exclude that from the crash report.

`excludeMatchingSharedPreferencesKeys` takes a list of regular expression patterns, following the regular expression syntax used by Java's `Pattern` class. If you do not use any `Pattern`-specific control characters, the default is basically a plain string match.

So, for example, if you have a `serverToken` `SharedPreferences` value that you would like to exclude, use:

```
excludeMatchingSharedPreferencesKeys={"serverToken"}
```

in your `@ReportsCrashes` annotation.

End-User Configuration

ACRA monitors certain default `SharedPreferences` values and configures its behavior based upon them. By exposing those preferences in your own `PreferenceFragment` or `PreferenceActivity`, you can allow the user to control ACRA's behavior.

The following table outlines the options:

Preference Key	Role	Data Type	Preference Type
`acra.disable`	Enable or disable ACRA reporting outright	`boolean`	`CheckBoxPreference`
`acra.syslog.enable`	Include Logcat data in crash reports	`boolean`	`CheckBoxPreference`
`acra.deviceid.enable`	Include device ID (e.g., IMEI) in crash reports	`boolean`	`CheckBoxPreference`
`acra.user.email`	Email address to include in reports	`String`	`EditTextPreference`
`acra.alwaysaccept`	If `true`, reports are always sent, even for dialog or notification modes	`boolean`	`CheckBoxPreference`

Note that `acra.disable` has an `acra.enable` counterpart. Only use one of these. A value of `true` for `acra.disable` is equivalent to `false` for `acra.enable`.

Also note that `acra.syslog.enable` is less important nowadays. In earlier versions of Android, where apps could get at everything in Logcat, it is possible that including Logcat data might have privacy implications for users, since that data would include logging from other apps. Since Android 4.1 no longer allows apps to get at Logcat data from other apps, this particular concern is moot.

ACRA and Processes

If you crash with an unhandled exception, your existing process is in a state of disarray. To help ensure that ACRA can report your crash successfully, ACRA by default runs its error-reporting `Service` in a separate ACRA-specific process.

Most of the time, this does not affect you.

One place where it might is if you have a custom `Application` subclass for things beyond ACRA. Bear in mind that each process gets its own copy of the virtual machine (Dalvik or ART) and each gets its own independent instance of your designated `Application` class. Work that you do for your main app's process you may wish to skip for the ACRA-specific background process.

To handle this, you can call `isACRASenderServiceProcess()` on the `ACRA` class. If this returns `true`, your `Application` is being created for ACRA's use, and you can skip any custom initialization work. If this returns `false`, this is one of your processes, and you can proceed normally.

Note that `isACRASenderServiceProcess()` was added to ACRA 4.9.0.

Trail: Miscellaneous Topics

In-App Diagnostics

Android has many tools to help you make sense of what is going on in your app, from complex tools like Traceview to simpler things like Logcat. Plus, if you are using an IDE, you have access to a debugger, which can let you step through code, inspect data members and other variables, and so on.

However, they all have one element in common: they are general-purpose tools. They know nothing specifically about your app, just Android apps in general. As a result, there may be information that *you* can gather that would be of immense benefit for debugging and diagnostic purposes, but that the general-purpose tools cannot collect for you.

More importantly, you need some way to see any diagnostic data that you collect. Logging stuff to Logcat can sometimes work, but then you have to worry about accidentally shipping that logging code in production, which would be less than ideal. And there are many cases where Logcat itself will not be a great visualization of the information.

What would be better is if we could add our own diagnostic tools to our app, for use while debugging, while excluding them from our release builds. And it would be great if we could add in these tools without changing much, if anything, of our production code to reference them. This chapter will explore how to implement such tools.

Prerequisites

In addition to the core chapters, it would be a good idea if you had read:

- the chapter on Gradle and the new project structure, to understand more about the `debug` source set
- the chapter on manifest merging, to understand how the `debug` source set can contribute to the overall app's manifest

Also, one of the techniques bears some resemblance to the tapjacking attack, though fortunately without the privacy and security ramifications.

One of the sample apps is based on a `RecyclerView` sample app, and so you may wish to skim the `RecyclerView` material to ensure that you understand enough about what is going on with the sample.

One of the sample apps uses an embedded Web server, based on concepts and a module covered in another chapter.

The Diagnostic Activity

Having a "back door" to get at diagnostic information about a program is a time-honored technique. Alas, far too many of those back doors wind up in production code, and too many of those wind up resulting in privacy or security flaws. Yet, the approach is still used to this day.

From a GUI standpoint, these back doors usually required some sort of special key sequence to initiate (e.g., press Ctrl-Shift-Z three times in less than a second). The objective was to make them easy enough to get to but not something that would routinely get in the way. And, for those back doors that wound up shipping, eventually word would get out about the magic key sequence, leading to all sorts of trouble.

In Android, we can dispense with the magic key sequence (which is good, since we often are not using keyboards). An app can have as many launcher icons as it needs, so we just need a launcher icon to get into some custom diagnostic activity that we want. However, now we *really* do not want to ship this code in production, as the diagnostic activity is no longer hidden, but rather is in plain view in the user's home screen launcher.

Fortunately, the advent of source sets with the Android Gradle Plugin, plus a reasonably robust manifest merger process, makes setting up this sort of tool fairly easy, yet keeps it out of the production code. Most of the work will be in actually writing the activity to report on whatever it is that you wanted reported on.

The Diagnostics/Activity. sample project will illustrate this process.

This app is a clone of a previous sample that retrieves Stack Overflow questions in the android tag via Square's Retrofit library. It also uses Square's Picasso library to load in the avatars of the people asking the questions. Picasso has an API for getting at statistics about the images that were downloaded: how many, how big, how many were already cached, and so on. The revised sample shown in this section will create a diagnostic activity that reports this information, as an illustration of having such an activity supply statistics that may be useful in tuning, debugging, etc.

The Sourceset

This project has two source sets, main and debug. main is where the production code lies; debug is where the diagnostic activity resides. The debug source set is tied to the debug build type, so only when doing a debug build will our debug code be included in the app. Since your production signing key is (hopefully) only being used by your release build type, this helps ensure that the diagnostic code does not ship with your production app.

The Manifest

Both source sets have manifests. For debug builds, the debug source set's manifest will be merged with the main source set's manifest to create the combined result.

The objective is to have the debug source set's manifest have the minimum elements and attributes required to have it successfully add what it needs to the app. The more stuff in a source set's manifest, the more likely it is that the stuff will conflict with similar stuff from main or other manifests and cause build problems.

Here, the debug manifest simply declares a new <activity>:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

  <application>
    <activity
      android:name="com.commonsware.android.debug.util.PicassoDiagnosticActivity"
      android:label="@string/picasso_diagnostics"
      android:taskAffinity="com.commonsware.android.debug.activity.PicassoDiagnosticActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```

```
  </application>

</manifest>
```

(from Diagnostics/Activity/app/src/debug/AndroidManifest.xml)

Note that the class name for the `PicassoDiagnosticActivity` is fully-qualified (`com.commonsware.android.debug.util.PicassoDiagnosticActivity`). For the purposes of this particular diagnostic, the activity does not have to be in the same package as the rest of the app. In fact, this activity could be in a library that could be referenced by many apps, if desired.

Also note the `taskAffinity` for the `<activity>` is set to its fully-qualified class name. This helps ensure that this activity will reside in a different task than does our main UI, so that the diagnostics activity does not artificially alter BACK button processing and the like from the regular task.

Since the `main` source set will not contain this particular `<activity>` element, there are no collisions, and the manifest merger will turn out clean.

The Activity

The activity itself is rather boring.

It loads in a layout resource containing a `TableLayout` that will contain our Picasso diagnostic report:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:shrinkColumns="1"
    android:stretchColumns="1">

    <TableRow>

      <TextView
        android:id="@+id/last_updated"
        style="@style/TableText.Title"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Last Updated"/>

    <TextView
      android:id="@+id/last_updated_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/avg_download_size"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Average Download Size"/>

    <TextView
      android:id="@+id/avg_download_size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/avg_orig_size"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Average Original Bitmap Size"/>

    <TextView
      android:id="@+id/avg_orig_size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/avg_xform_size"
      style="@style/TableText.Title"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Average Transformed Bitmap Size"/>

    <TextView
      android:id="@+id/avg_xform_size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/cache_hits"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Cache Hits"/>

    <TextView
      android:id="@+id/cache_hits_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/cache_misses"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Cache Misses"/>

    <TextView
      android:id="@+id/cache_misses_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/download_count"
      style="@style/TableText.Title"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Download Count"/>

    <TextView
      android:id="@+id/download_count_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/max_size"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Max Size"/>

    <TextView
      android:id="@+id/max_size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/orig_bitmap_count"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Original Bitmap Count"/>

    <TextView
      android:id="@+id/orig_bitmap_count_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/size"
      style="@style/TableText.Title"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Size"/>

    <TextView
      android:id="@+id/size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/total_dl_size"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Total Download Size"/>

    <TextView
      android:id="@+id/total_dl_size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/total_orig_size"
      style="@style/TableText.Title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Total Original Size"/>

    <TextView
      android:id="@+id/total_orig_size_value"
      style="@style/TableText.Value"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView
      android:id="@+id/total_xform_size"
      style="@style/TableText.Title"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Total Transformed Size"/>

      <TextView
        android:id="@+id/total_xform_size_value"
        style="@style/TableText.Value"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    </TableRow>

    <TableRow>

      <TextView
        android:id="@+id/xform_count"
        style="@style/TableText.Title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Transformed Count"/>

      <TextView
        android:id="@+id/xform_count_value"
        style="@style/TableText.Value"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    </TableRow>
  </TableLayout>
</ScrollView>
```

(from Diagnostics/Activity/app/src/debug/res/layout/main.xml)

That layout, in turn, references some custom styles, to avoid having to repeat the configuration of each of the `TextView` widgets quite so much:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="TableText">
    <item name="android:textSize">12sp</item>
    <item name="android:layout_margin">8dp</item>
  </style>

  <style name="TableText.Title">
    <item name="android:textStyle">bold</item>
  </style>

  <style name="TableText.Value">
    <item name="android:typeface">monospace</item>
```

```
  </style>
</resources>
```

(from Diagnostics/Activity/app/src/debug/res/values/styles.xml)

The activity loads the layout, gets a `StatsSnapshot` from Picasso containing a snapshot of the results of using Picasso, and pours the data into the various `TextView` widgets:

```
package com.commonsware.android.debug.util;

import android.app.Activity;
import android.os.Bundle;
import android.text.format.DateUtils;
import android.widget.TextView;
import com.commonsware.android.debug.activity.R;
import com.squareup.picasso.Picasso;
import com.squareup.picasso.StatsSnapshot;

public class PicassoDiagnosticActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    StatsSnapshot ss=Picasso.with(this).getSnapshot();

    TextView tv=(TextView)findViewById(R.id.last_updated_value);

    tv.setText(DateUtils.formatDateTime(this, ss.timeStamp,
                                        DateUtils.FORMAT_SHOW_TIME));

    tv=(TextView)findViewById(R.id.avg_download_size_value);
    tv.setText(Long.toString(ss.averageDownloadSize));

    tv=(TextView)findViewById(R.id.avg_orig_size_value);
    tv.setText(Long.toString(ss.averageOriginalBitmapSize));

    tv=(TextView)findViewById(R.id.avg_xform_size_value);
    tv.setText(Long.toString(ss.averageTransformedBitmapSize));

    tv=(TextView)findViewById(R.id.cache_hits_value);
    tv.setText(Long.toString(ss.cacheHits));

    tv=(TextView)findViewById(R.id.cache_misses_value);
    tv.setText(Long.toString(ss.cacheMisses));
```

```
    tv=(TextView)findViewById(R.id.download_count_value);
    tv.setText(Long.toString(ss.downloadCount));

    tv=(TextView)findViewById(R.id.max_size_value);
    tv.setText(Long.toString(ss.maxSize));

    tv=(TextView)findViewById(R.id.orig_bitmap_count_value);
    tv.setText(Long.toString(ss.originalBitmapCount));

    tv=(TextView)findViewById(R.id.size_value);
    tv.setText(Long.toString(ss.size));

    tv=(TextView)findViewById(R.id.total_dl_size_value);
    tv.setText(Long.toString(ss.totalDownloadSize));

    tv=(TextView)findViewById(R.id.total_orig_size_value);
    tv.setText(Long.toString(ss.totalOriginalBitmapSize));

    tv=(TextView)findViewById(R.id.total_xform_size_value);
    tv.setText(Long.toString(ss.totalTransformedBitmapSize));

    tv=(TextView)findViewById(R.id.xform_count_value);
    tv.setText(Long.toString(ss.transformedBitmapCount));
  }
}
```

(from Diagnostics/Activity/app/src/debug/java/com/commonsware/android/debug/util/PicassoDiagnosticActivity.java)

The Results

If you install the app on a device or emulator from a `debug` build, you will get two launcher icons. The one labeled "Picasso Diagnostics" will be the `PicassoDiagnosticsActivity`. If you bring up that activity after having run the main activity, you will see some information about the images that Picasso loaded:

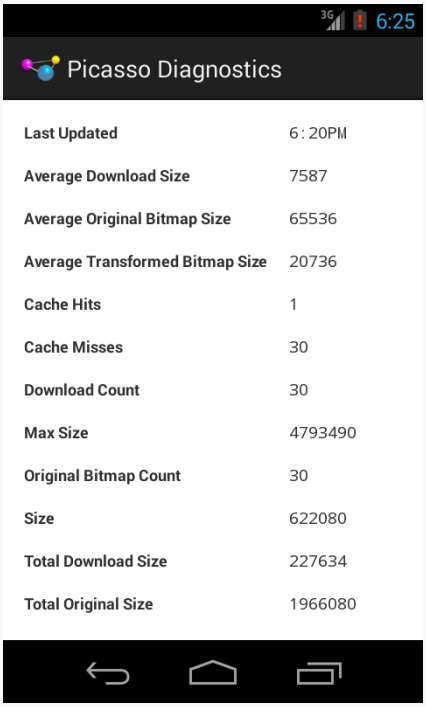


Figure 1068: Picasso Diagnostic Activity

A `release` build, on the other hand, does not include the extra activity, its resources, or its manifest entry, since those are all in the `debug` source set.

Also, nothing from this affected our `main` source set contents. We did not have to add things to the manifest, or adjust our Java code, or anything of the sort.

The Limitations

While this sample is fairly trivial, these sorts of diagnostic activities can be as elaborate as is needed. In some cases, as with this sample, the results are reusable — so long as the app has Picasso, this code can add in the diagnostic activity.

However, this is only good for post-mortem sorts of diagnostics, where you do something in the "real" app, then head over to the diagnostic activity to see what it has to report. In many cases, this is perfectly reasonable. In other cases, the act of switching to the diagnostic activity might affect the diagnostics, if those diagnostics are dependent upon things like activity lifecycle methods. You also cannot learn anything in real time, seeing both the app and the diagnostics simultaneously (or nearly so).

However, there are other options that can improve in these areas, for situations that need such improvement.

The Diagnostic Web App

If switching the content of our device's screen impedes our ability to use the diagnostic information, is there a way we could get another screen involved?

One likely screen would be your development machine (desktop, notebook, etc.). After all, you can get that with Logcat messages. However, Logcat messages are limited in terms of formatting and rendering control. What would be interesting is if we could get information from our app to the development machine, in real time, other than via Logcat.

For example, we could have a Web app embedded in our Android app, to allow us to serve up Web pages to our development machine.

The What, Now?

We are used to having Web apps, but usually running on Web servers, whether shared hosts or cloud providers (e.g., Amazon's AWS) or that Web server box you have tucked in the corner of the office.

There is nothing stopping you from embedding a Web server in an Android app. After all, Android has access to Java's `ServerSocket`, just as Java EE servers do. It is merely a matter of having some HTTP server stack that is small enough to be embeddable but reasonably robust for whatever the targeted use is. In our case, the "targeted use" is being able to provide us with diagnostic data, which means we are not aiming for lots of simultaneous users, terabyte-scale data stores, or any of the typical things that you see "serious" Web developers have to deal with.

On the other hand, most "serious" Web developers do not have to worry about their Web servers hooking up to the WiFi in a coffee shop, either.

The Security Ramifications

Having any sort of open port on an Android device is a scary proposition from a security standpoint. At minimum, anyone on the same WiFi LAN segment can access that port. Some mobile carriers assign public IP addresses to devices on their networks, and in those cases, *anyone* can access that port.

In our case, we will only be using this Web server for debug builds, which limits the harm that can be caused. Now the only people who might be attacked are... ourselves.

When using this technique, you will also want to ensure that you have adequate controls to be able to stop the server, so it is only open and accepting requests when you want it to.

The Sample App

The `Diagnostics/WebServer`. sample project is based off of the previous project, where we are trying to get diagnostic information about Picasso's operation and image cache. But, whereas the preceding project used a dedicated activity, this app will surface this information via an embedded Web server.

Once again, we will use the `debug` source set to minimize the amount of code changes needed to the `main` source set code and so we can feel confident that our Web server will not sneak into our production app somehow.

However, while Android has support for activities built in, it does not have a Web server built in, nor much in the way of a modern template engine for generating dynamic Web content. For those, we will turn to third parties.

The chapter on embedding a Web server covers the custom Web server implementation that we will be extending. If you have not read that chapter yet, probably you should do so before continuing. This section will not make much sense otherwise.

Adding the Library Module

This sample relies upon the reusable Web server module outlined in the chapter on embedding a Web server in your Android app.

This sample references the reusable Web server module directly, rather than via an AAR artifact, by hacking the relative path into `settings.gradle` for the project:

```
include ':app', ':webserver'

project(':webserver').projectDir=new File('../../WebServer/Reusable/webserver')
```

(from Diagnostics/WebServer/settings.gradle)

The app module, in turn, has some changes in its `build.gradle` file as well.

First, we have a dependency on the `:webserver` module added to `settings.gradle`. And, for whatever reason, Gradle and Android Studio are happier if we duplicate the `support-v13` dependency, which is already being pulled in via the dependency on `:webserver`:

```
dependencies {
    implementation 'com.squareup.picasso:picasso:2.5.2'
    implementation 'com.squareup.retrofit:retrofit:1.9.0'
    debugCompile project(':webserver')
    debugCompile 'com.android.support:support-v13:25.0.3'
}
```

(from Diagnostics/WebServer/app/build.gradle)

Also, Android's build tools will want to compress HTML and JavaScript assets, even though that will interfere with our serving them. Hence, in the `android` closure, we have an `aaptOptions` closure that specifically requests to not compress those assets:

```
    aaptOptions {
        noCompress 'html', 'js'
    }
```

(from Diagnostics/WebServer/app/build.gradle)

The Web Content

This app will use Handlebars.java as a template engine, since that is part of our resuable Web server module. It would help to have some actual templates. In our case, we really only need one, one that will show the Picasso diagnostic output. So, the sample app has a 1995-era Web page (sans `<blink>` tag, though) that has a simple table showing those Picasso values:

```
<html>
<head><title>Picasso Diagnostics</title></head>
<body>
<h1>Picasso Diagnostics</h1>
<table>
<tr><th style="text-align:right; padding-right:16px;">Timestamp</th><td>{{timeStamp}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Average Download
Size</th><td>{{averageDownloadSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Average Original Bitmap
Size</th><td>{{averageOriginalBitmapSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Average Transformed Bitmap
Size</th><td>{{averageTransformedBitmapSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Cache Hits</th><td>{{cacheHits}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Cache Misses</th><td>{{cacheMisses}}</td></tr>
```

```
<tr><th style="text-align:right; padding-right:16px;">Download Count</th><td>{{downloadCount}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Max Size</th><td>{{maxSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Original Bitmap
Count</th><td>{{originalBitmapCount}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Size</th><td>{{size}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Total Download
Size</th><td>{{totalDownloadSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Total Original Bitmap
Size</th><td>{{totalOriginalBitmapSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Total Transformed Bitmap
Size</th><td>{{totalTransformedBitmapSize}}</td></tr>
<tr><th style="text-align:right; padding-right:16px;">Transformed Bitmap
Count</th><td>{{transformedBitmapCount}}</td></tr>
</table>
<hr/>
<a href="/stop">Stop Service</a>
</body>
</html>
```

(from Diagnostics/WebServer/app/src/debug/assets/picasso.hbs)

The values themselves are represented as mustache-style value references (e.g., `{{cacheHits}}`). Those will be replaced at runtime by actual values pulled from Picasso and supplied to Handlebars.java.

This template resides in `assets/` of the `debug` source set. That means that the template will not ship with the production app.

PicassoDiagnosticService

`PicassoDiagnosticService` is a subclass of the reusable `WebServerService` that will publish our Picasso diagnostic information to Web browsers.

Many of the methods in `PicassoDiagnosticService` are there to satisfy `WebServerService` and its `abstract` methods. So, our `getPort()`, `getMaxIdleTimeSeconds()`, and `getMaxSequentialInvalidRequests()` methods are fairly rote:

```
  @Override
  protected int getPort() {
    return(4999);
  }

  @Override
  protected int getMaxIdleTimeSeconds() {
    return(120);
  }

  @Override
```

```
protected int getMaxSequentialInvalidRequests() {
  return(10);
}
```

(from Diagnostics/WebServer/app/src/debug/java/com/commonsware/android/debug/webserver/PicassoDiagnosticService.java)

Similarly, `buildForegroundNotification()` just sets up our `Notification` with a pointer back to `PicassoDiagnosticActivity`, the activity that we will use to start and stop the server:

```
@Override
protected void buildForegroundNotification(NotificationCompat.Builder b) {
  Intent iActivity=new Intent(this, PicassoDiagnosticActivity.class);
  PendingIntent piActivity=PendingIntent.getActivity(this, 0,
    iActivity, 0);

  b.setContentTitle(getString(R.string.app_name))
    .setContentIntent(piActivity)
    .setSmallIcon(R.drawable.ic_launcher)
    .setTicker(getString(R.string.app_name));
}
```

(from Diagnostics/WebServer/app/src/debug/java/com/commonsware/android/debug/webserver/PicassoDiagnosticService.java)

In addition to stopping the server from `PicassoDiagnosticActivity`, it might be nice to be able to stop it from a link in the Web page being served up. So, that page has a link to a `stop` relative path, and we set up that route in `configureRoutes()`:

```
@Override
protected boolean configureRoutes(AsyncHttpServer server) {
  server.get("/stop", new StopRequestCallback());

  return(true);
}
```

(from Diagnostics/WebServer/app/src/debug/java/com/commonsware/android/debug/webserver/PicassoDiagnosticService.java)

Returning `true` tells `WebServerService` to handle standard serve-content-from-assets logic itself, which we will be using for serving up a Web page based on the Handlebars template.

`StopRequestCallback` is a fairly trivial `HttpServerRequestCallback` implementation, just sending back some stub content and calling `stopSelf()` to stop the service (and, thereby, tear down the Web server):

```
private class StopRequestCallback implements HttpServerRequestCallback {
  @Override
  public void onRequest(AsyncHttpServerRequest request, AsyncHttpServerResponse response) {
    response.send("Goodbye, cruel world!");
    stopSelf();
  }
}
```

(from Diagnostics/WebServer/app/src/debug/java/com/commonsware/android/debug/webserver/PicassoDiagnosticService.java)

Most of the business logic for this server lies in `getContextForPath()`. This will be called on our service whenever one of our Handlebars templates is requested, so we can provide the Handlebars `Context` (not to be confused with an Android `Context`). The `Context` is used to fill in all of the macros used in the template.

In our case, most of the macros are pulling from the Picasso `StatsSnapshot`, so we need to make sure that our `Context` has it. However, it would be nice to include the timestamp of when the snapshot was taken, formatted in a more conventional format than "milliseconds since the Unix epoch". While `StatsSnapshot` has the timestamp, we need to do the formatting.

So, `getPathForContext()` gets the `StatsSnapshot` and the formatted timestamp and combines them into a suitable `Context`:

```
@Override
protected Context getContextForPath(String relpath) {
  if ("picasso.hbs".equals(relpath)) {
    StatsSnapshot ss=Picasso
      .with(PicassoDiagnosticService.this)
      .getSnapshot();
    String formattedTime=DateUtils.formatDateTime(PicassoDiagnosticService.this,
      ss.timeStamp,
      DateUtils.FORMAT_SHOW_TIME);

    return(Context
      .newBuilder(ss)
      .combine("formattedTime", formattedTime)
      .resolver(FieldValueResolver.INSTANCE)
      .build());
  }

  throw new IllegalStateException("Did not recognize "+relpath);
}
```

(from Diagnostics/WebServer/app/src/debug/java/com/commonsware/android/debug/webserver/PicassoDiagnosticService.java)

Here, we are saying that the core data comes from the `StatsSnapshot` (`newBuilder(ss)`), where macros are interpreted as fields on the object (`resolver(FieldValueResolver.INSTANCE)`). But, if anyone asks for `formattedTime`, we supply that separately (`combine("formattedTime", formattedTime)`).

Note that our manifest has our service, including the action string used by the reusable library to be able to stop our service from the foreground `Notification`:

```
    <service
      android:name="PicassoDiagnosticService"
      android:exported="false">
      <intent-filter>
        <action android:name="com.commonsware.android.webserver.WEB_SERVER_SERVICE"/>
      </intent-filter>
    </service>
```

(from Diagnostics/WebServer/app/src/debug/AndroidManifest.xml)

The Launcher Activity

We need to arrange to start the `PicassoDiagnosticService` at some point.

One possibility would be to add code to the main app that checked to see if we were running a `debug` build and would start the service then. On the plus side, it would mean that the service would be running all the time, without additional work on the part of the developer. However:

- The service itself will affect the behavior of the app, particularly if we have no other services, such as by keeping the process around longer than normal, and this might affect our testing
- This approach would require modifying the `main` source set to be aware of the `debug` code, which is not ideal

Instead, we use a clone of the activity used in the other sample apps in the chapter on the embedded Web server:

```
package com.commonsware.android.debug.webserver;

import android.app.ListActivity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import com.commonsware.android.webserver.WebServerService;
import java.util.ArrayList;
import de.greenrobot.event.EventBus;

public class PicassoDiagnosticActivity extends ListActivity {
  private MenuItem record, stop;

  @Override
```

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
}

@Override
protected void onStart() {
  super.onStart();

  EventBus.getDefault().registerSticky(this);
}

@Override
protected void onStop() {
  EventBus.getDefault().unregister(this);

  super.onStop();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  record=menu.findItem(R.id.record);
  stop=menu.findItem(R.id.stop);

  WebServerService.ServerStartedEvent event=
    EventBus.getDefault().getStickyEvent(WebServerService.ServerStartedEvent.class);

  if (event!=null) {
    onEventMainThread(event);
  }

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  Intent i=new Intent(this, PicassoDiagnosticService.class);

  if (item.getItemId()==R.id.record) {
    startService(i);
  }
  else {
    stopService(i);
  }

  return super.onOptionsItemSelected(item);
}

@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
  startActivity(new Intent(Intent.ACTION_VIEW,
    Uri.parse(getListAdapter().getItem(position).toString())));
}

public void onEventMainThread(WebServerService.ServerStartedEvent event) {
  if (record!=null) {
    record.setVisible(false);
    stop.setVisible(true);
```

```
      ArrayList<String> diagUrls=new ArrayList<String>();

      for (String url : event.getUrls()) {
        diagUrls.add(url+"picasso.hbs");
      }

      setListAdapter(new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, diagUrls));
    }
  }

  public void onEventMainThread(WebServerService.ServerStoppedEvent event) {
    if (record!=null) {
      record.setVisible(true);
      stop.setVisible(false);
      setListAdapter(null);
    }
  }
}
```

(from Diagnostics/WebServer/app/src/debug/java/com/commonsware/android/debug/webserver/PicassoDiagnosticActivity.java)

Beyond updating class names to refer to our classes instead of ones from the other samples, the only change of significance comes in setting up the URLs for the `ListView`. Here, we append the `picasso.hbs` portion onto the base URL, so tapping the `ListView` row will bring up the desired Web page containing our report.

The Diagnostic Overlay

Sometimes, the information that we want needs to be presented to the developer in real time, while the user is looking at the UI of the app.

Take `StrictMode`, for example.

`penaltyDeath()` can be used to totally crash the app when, say, the app detects network I/O on the main application thread. However, this is rather harsh, particularly since Google ships all sorts of code in the framework that does improper things on the main application thread.

`penaltyLog()` is great, in that it gives us the stack trace of where the problem is but does not outright crash the app. However, we might not notice the stack traces, if we are not paying close attention to Logcat.

So, one popular combination is `penaltyFlashScreen()` combined with `penaltyLog()`. `penaltyFlashScreen()` will flash a red border around the edges of the screen, as a hint that "hey, something was detected – please check Logcat for

details!" In the absence of this, or some other visual penalty, like `penaltyDialog()`, the developer might not learn what `StrictMode` is trying to tell her.

Another example is `gfxinfo`. As is noted in the chapter on jank, `gfxinfo` can give you some information about how long it takes for frames to be rendered, so you know when you are dropping frames. One option for this is via an overlay that appears on top of the main UI, so you can see in real time a bar chart of frame times, so you can see what user actions can trigger jank.

You can employ the same sorts of techniques yourself, to put an overlay on the screen, whether temporarily (e.g., the `StrictMode` `penaltyFlashScreen()` option) or more durably (e.g., the `gfxinfo` bar chart). The `Diagnostics/Overlay`. sample project will demonstrate the former, alerting you of slowdowns in the rendering of your items in a `RecyclerView`.

The Gradle Setup

For the purposes of demonstration, we need a sample app that can actually perform poorly, so we detect slowdowns and alert the developer via the screen overlay. At the same time, we need a sample app that does *not* perform poorly, so we can determine if the overlay works properly in both cases.

This sample project toggles between the two modes via a custom `BE_STUPID` field added to the `BuildConfig` class, based upon build type:

```
    buildTypes {
        debug {
            buildConfigField "boolean", "BE_STUPID", "true"
        }

        release {
            buildConfigField "boolean", "BE_STUPID", "false"
        }
    }
```

(from Diagnostics/Overlay/app/build.gradle)

Admittedly, we could have skipped this and used the `BUILD_TYPE` field on `BuildConfig`. That will hold whatever the name of the build type was that built the APK that we installed, so it will be `debug` or `release` in this project. However, in case you wanted to have different rules for when the app should be stupid or not, we pull it out into a separate `BuildConfig` field. For example, you might elect to add two

product flavors, so switching between `true` and `false` for `BE_STUPID` is based on product flavor instead of build type.

Introducing RVAdapterWrapper

The wrapper pattern in Java can be fairly powerful, allowing you to extend Java objects without changing inheritance hierarchies. Rather, you wrap the object in a wrapper that implements the same interface (or inherits from the same base class). The wrapper can do some things on its own (the extended behavior) and delegate to the wrapped object for everything else. Plus, *subclasses* of the wrapper can basically override stock wrapper behavior and behavior the wrapped object.

Android has a few such wrapper classes, like `CursorWrapper` and `ContextWrapper`. The author of this book published an `AdapterWrapper`, for the `AdapterView` family of classes, as a tiny open source library. And this chapter has `RVAdapterWrapper`, which implements a wrapper for `RecyclerView.Adapter`:

```
package com.commonsware.android.debug.videolist;

import android.support.v7.widget.RecyclerView;
import android.view.ViewGroup;

public class RVAdapterWrapper<T extends RecyclerView.ViewHolder> extends RecyclerView.Adapter<T> {
  private final RecyclerView.Adapter<T> wrapped;

  public RVAdapterWrapper(RecyclerView.Adapter<T> wrapped) {
    super();

    this.wrapped=wrapped;
  }

  public RecyclerView.Adapter<T> getWrappedAdapter() {
    return(wrapped);
  }

  @Override
  public T onCreateViewHolder(final ViewGroup parent, final int viewType) {
   return(wrapped.onCreateViewHolder(parent, viewType));
  }

  @Override
  public void onBindViewHolder(final T holder, final int position) {
    wrapped.onBindViewHolder(holder, position);
  }

  @Override
  public long getItemId(int position) {
    return(wrapped.getItemId(position));
  }

  @Override
  public int getItemViewType(int position) {
```

```
    return(wrapped.getItemViewType(position));
  }

  @Override
  public void onAttachedToRecyclerView(RecyclerView recyclerView) {
    wrapped.onAttachedToRecyclerView(recyclerView);
  }

  @Override
  public void onDetachedFromRecyclerView(RecyclerView recyclerView) {
    wrapped.onDetachedFromRecyclerView(recyclerView);
  }

  @Override
  public void onViewAttachedToWindow(T holder) {
    wrapped.onViewAttachedToWindow(holder);
  }

  @Override
  public void onViewDetachedFromWindow(T holder) {
    wrapped.onViewDetachedFromWindow(holder);
  }

  @Override
  public void onViewRecycled(T holder) {
    wrapped.onViewRecycled(holder);
  }

  @Override
  public void registerAdapterDataObserver(RecyclerView.AdapterDataObserver observer) {
    wrapped.registerAdapterDataObserver(observer);
  }

  @Override
  public void setHasStableIds(boolean hasStableIds) {
    wrapped.setHasStableIds(hasStableIds);
  }

  @Override
  public void unregisterAdapterDataObserver(RecyclerView.AdapterDataObserver observer) {
    wrapped.unregisterAdapterDataObserver(observer);
  }

  @Override
  public int getItemCount() {
    return(wrapped.getItemCount());
  }
}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/RVAdapterWrapper.java)

The constructor takes the `RecyclerView.Adapter` to be wrapped, and `RVAdapterWrapper` offers a `getWrappedAdapter()` to retrieve that object. Everything else is a simple implementation of the wrapper pattern, overriding all methods from `RecyclerView.Adapter` and delegating them to the wrapped adapter.

TimingWrapper (a.k.a., StrictMode for RecyclerView)

`RVAdapterWrapper` exists mostly to serve as a base class for `TimingWrapper`. `TimingWrapper` arranges to collect the time used by `onCreateViewHolder()` and `onBindViewHolder()`, so we can see if those times exceed some threshold and therefore is worthy of alerting the developer.

Its constructor takes the `RecyclerView.Adapter` to wrap, along with the `Activity` that is hosting this UI. In addition to chaining to the superclass and holding onto that `Activity` (as a data member named `host`), the constructor also retrieves a `WindowManager` system service:

```
public TimingWrapper(RecyclerView.Adapter<T> wrapped, Activity host) {
  super(wrapped);

  this.host=host;
  wm=(WindowManager)host.getSystemService(Context.WINDOW_SERVICE);
}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/TimingWrapper.java)

`TimingWrapper` overrides `onCreateViewHolder()` and `onBindViewHolder()`, tracking the amount of time that those calls take, and calling a private `warn()` method with the time for the call:

```
@Override
public T onCreateViewHolder(final ViewGroup parent, final int viewType) {
  long start=SystemClock.uptimeMillis();
  T result=super.onCreateViewHolder(parent, viewType);

  warn(SystemClock.uptimeMillis() - start);

  return(result);
}

@Override
public void onBindViewHolder(final T holder, final int position) {
  long start=SystemClock.uptimeMillis();

  super.onBindViewHolder(holder, position);
  warn(SystemClock.uptimeMillis() - start);
}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/TimingWrapper.java)

Here, `T` is the `RecyclerView.ViewHolder` used by the wrapped adapter and declared by the specific use of the `TimingAdapter`:

```
public class TimingWrapper<T extends RecyclerView.ViewHolder> extends RVAdapterWrapper<T> {
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/TimingWrapper.java)

`warn()` sees if the amount of time the call took exceeds some threshold, set here to be 7ms. If it does, we first log a stack trace to Logcat, following the technique used by `StrictMode` itself of having a private `LogStackTrace Exception` that is just there to collect a stack trace:

```
  private void warn(long delta) {
    if (delta>7) {
      String msg=String.format("RVAdapterWrapper violation: ~duration= %d ms",
          delta);

      Log.e(TAG, msg, new LogStackTrace());

      if (v==null) {
        WindowManager.LayoutParams params=new WindowManager.LayoutParams(
            WindowManager.LayoutParams.MATCH_PARENT,
            WindowManager.LayoutParams.MATCH_PARENT,
            WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,
            WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
                | WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE,
            PixelFormat.TRANSLUCENT);

        v=new View(host);
        v.setBackgroundResource(R.drawable.border);
        wm.addView(v, params);

        v.postDelayed(new Runnable() {
          @Override
          public void run() {
            wm.removeView(v);
            v=null;
          }
        }, 500);
      }
    }
  }

  private static class LogStackTrace extends Exception {}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/TimingWrapper.java)

Then, if we are not presently showing an overlay (i.e., the `v` data member is not `null`), we:

- create a `WindowManager.LayoutParams` that will fill the screen, using `TYPE_SYSTEM_OVERLAY` as the type, and indicating that it has a translucent background
- create a simple `View` and give it a background defined as the `border` drawable resource
- use the `WindowManager` to show that view
- use `postDelayed()` to get control in 500ms and remove that view, also setting `v` back to `null`

`border` is defined in `res/drawable-nodpi/` as a `ShapeDrawable`, consisting of a transparent rectangle with a 16dp-wide green border:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
       android:shape="rectangle">
  <stroke android:color="#ff00ff00" android:width="@dimen/border_width"/>
</shape>
```

(from Diagnostics/Overlay/app/src/main/res/drawable-nodpi/border.xml)

The net effect is that the border will flash a 16dp-wide green line around the edge of the screen for 500ms before being removed.

Note that to use the `TYPE_SYSTEM_OVERLAY`, we have to hold the `SYSTEM_ALERT_WINDOW` permission:

```
  <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
```

(from Diagnostics/Overlay/app/src/main/AndroidManifest.xml)

We will get more into the ramifications of that permission on Android 6.0+ devices later in this chapter.

The RecyclerViewActivity

You will notice that the `RVAdapterWrapper` and `TimingWrapper` code is in the `main` source set. Hence, those classes will exist on `debug` and `release` builds. However, we only *use* them on some builds, courtesy of a tweaked `RecyclerViewActivity`. The revised activity has `getAdapter()` and `setAdapter()` implementations that work with an unwrapped adapter, but hold onto a wrapped adapter, on `DEBUG` builds:

```
package com.commonsware.android.debug.videolist;

import android.app.Activity;
import android.os.Build;
import android.provider.Settings;
import android.support.v7.widget.RecyclerView;

public class RecyclerViewActivity extends Activity {
  private RecyclerView rv=null;

  public void setAdapter(RecyclerView.Adapter adapter) {
    boolean canDrawOverlays=
        (Build.VERSION.SDK_INT<=Build.VERSION_CODES.LOLLIPOP_MR1);

    if (!canDrawOverlays) {
      canDrawOverlays=Settings.canDrawOverlays(this);
    }

    if (BuildConfig.DEBUG && canDrawOverlays) {
      adapter=new TimingWrapper(adapter, this);
    }

    getRecyclerView().setAdapter(adapter);
  }

  public RecyclerView.Adapter getAdapter() {
    RecyclerView.Adapter result=getRecyclerView().getAdapter();

    if (result instanceof RVAdapterWrapper) {
      result=((RVAdapterWrapper)result).getWrappedAdapter();
    }

    return(result);
  }

  public void setLayoutManager(RecyclerView.LayoutManager mgr) {
    getRecyclerView().setLayoutManager(mgr);
  }

  public RecyclerView getRecyclerView() {
    if (rv==null) {
      rv=new RecyclerView(this);
      setContentView(rv);
    }

    return(rv);
  }
}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/RecyclerViewActivity.java)

One criterion for whether we will use the `TimingWrapper` is whether or not we are on a debug build. The other criteria take a bit more explanation, which we will get to later in this chapter.

Being Stupid

The overall sample app is a clone of a `RecyclerView` sample that loads videos from `MediaStore` and shows them in alphabetical order, along with thumbnails of the videos. This version of the sample is augmented with Advanced Be-Stupid Technology™, where `RowController` does the thumbnail retrieval on the main application thread when being stupid or uses Picasso when not:

```
void bindModel(Cursor row) {
  title.setText(row.getString(row.getColumnIndex(MediaStore.Video.Media.TITLE)));

  int uriColumn=row.getColumnIndex(MediaStore.Video.Media.DATA);
  int mimeTypeColumn=
      row.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);
  int videoId=row.getInt(row.getColumnIndex(MediaStore.Video.Media._ID));

  videoUri=row.getString(uriColumn);
  videoMimeType=row.getString(mimeTypeColumn);

  if (BuildConfig.BE_STUPID) {
    ContentResolver cr=thumbnail.getContext().getContentResolver();
    BitmapFactory.Options options=new BitmapFactory.Options();

    options.inSampleSize = 1;

    Bitmap thumb=MediaStore.Video.Thumbnails.getThumbnail(cr, videoId,
        MediaStore.Video.Thumbnails.MICRO_KIND, options);

    thumbnail.setImageBitmap(thumb);
  }
  else {
    Uri video=
        ContentUris.withAppendedId(MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
            videoId);

    Picasso.with(thumbnail.getContext())
      .load(video.toString())
      .fit().centerCrop()
      .placeholder(R.drawable.ic_media_video_poster)
      .into(thumbnail);
  }
}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/RowController.java)

The Results

The green border will flash if a call to `onCreateViewHolder()` or `onBindViewHolder()` takes more than 7ms. In theory, this should only occur if one of those methods does a non-trivial bit of work on the main application thread.

In practice, this seems to generate a fair number of false positives, presumably due to context-switching between threads on the available device cores.

Areas for Improvement

Those results point at areas where this technique might be improved:

- Pass the timing value into the `TimingWrapper` constructor, rather than hard-coding it to 7ms
- Rather than worrying about individual calls exceeding a 7ms threshold, point out if a rolling average of recent calls exceeds the threshold, to perhaps smooth out the data a bit and avoid the false positives

What Changed in Android 6.0

Let's go back to the `setAdapter()` implementation, where we conditionally apply the `TimingAdapter`:

```
public void setAdapter(RecyclerView.Adapter adapter) {
  boolean canDrawOverlays=
      (Build.VERSION.SDK_INT<=Build.VERSION_CODES.LOLLIPOP_MR1);

  if (!canDrawOverlays) {
    canDrawOverlays=Settings.canDrawOverlays(this);
  }

  if (BuildConfig.DEBUG && canDrawOverlays) {
    adapter=new TimingWrapper(adapter, this);
  }

  getRecyclerView().setAdapter(adapter);
}
```

(from Diagnostics/Overlay/app/src/main/java/com/commonsware/android/debug/videolist/RecyclerViewActivity.java)

As noted earlier, we only use `TimingWrapper` on debug builds, not release builds.

We also only use `TimingAdapter` if one of two things is true:

- Either we are on some version of Android prior to 6.0, or
- We are allowed to draw overlays

Historically, the `SYSTEM_ALERT_WINDOW` permission was merely listed as `dangerous`. Users would be notified about it at install time, but otherwise it was just a standard permission.

Originally, few apps requested this permission. Over time, more and more apps started using this for things like Facebook's "chatheads" UI.

In Android 6.0, `SYSTEM_ALERT_WINDOW` was moved to be a `signature`-level permission. Ordinarily, the net effect of this change would be that apps could no longer hold the permission, unless they were signed by the signing key that signed the firmware. While that's possible for device manufacturers and custom ROM developers, ordinary Android SDK developers would be left out.

However, Android 6.0 provided another means to get the rights to use `TYPE_SYSTEM_OVERLAY` windows, through a double-opt-in mechanism. The user not only has to install the app, but has to go to a particular screen in the Settings app to agree to grant your app the right to draw over top of other apps.

The default way for a user to get to that screen is to go into Settings > App, then click the gear icon in the action bar of the Settings app:

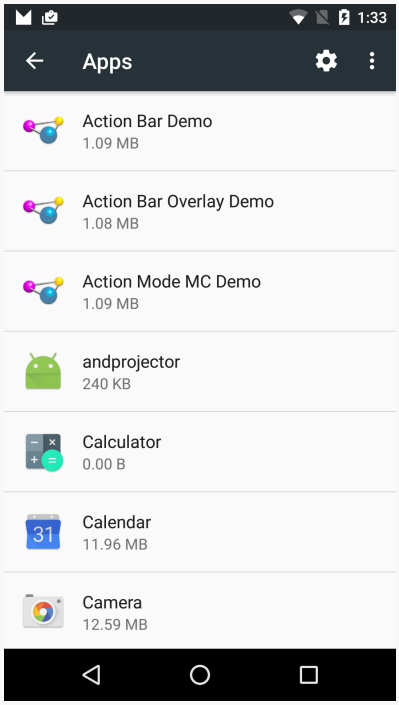


Figure 1069: Android 6.0 Settings App, Apps Screen, with Gear Icon

Tapping that brings up a “Configure apps” screen:

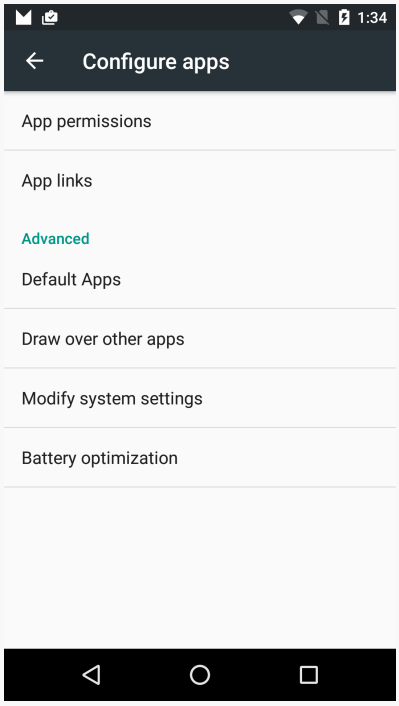


Figure 1070: Android 6.0 Settings App, Configure Apps Screen

There, tapping the "Draw over other apps" entry brings up a list of all of the apps that have requested the `SYSTEM_ALERT_WINDOW` permission:

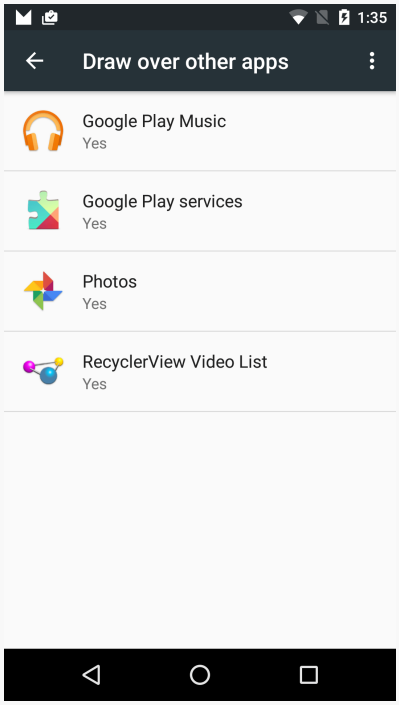


Figure 1071: Android 6.0 Settings App, Draw Over Other Apps Screen

Tapping on any one of those allows the user to toggle on or off this access:

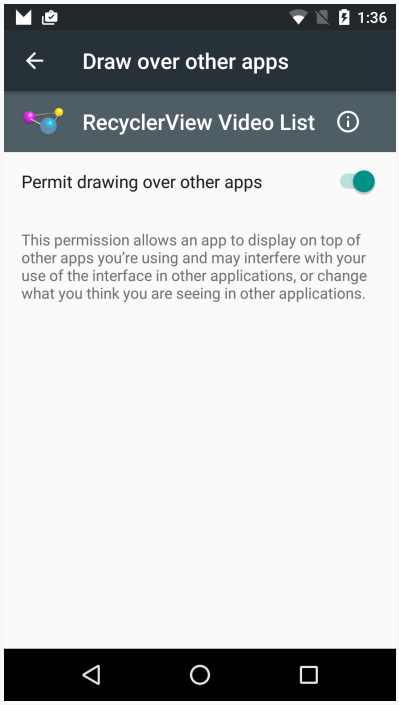


Figure 1072: Android 6.0 Settings App, Configuring Overlay Permission

In code, you can find out if the user has enabled this access by calling `canDrawOverlays()` on the `Settings` class, as we did in `setAdapter()` above. However:

- This requires you to have a `compileSdkVersion` of 23 or higher
- You cannot call that method on pre-Android 6.0 devices

On Android 6.0+, if `canDrawOverlays()` returns `false`, you are welcome to lead the user over to the appropriate screen in Settings to try to convince them to allow you to draw over other apps. To do that:

- Create a `package: Uri` that points to your app
- Wrap that in an `ACTION_MANAGE_OVERLAY_PERMISSION Intent`
- Call `startActivity()` to bring up that screen

```
Intent i=new Intent(Settings.ACTION_MANAGE_OVERLAY_PERMISSION,
                    Uri.parse("package:" + getPackageName()));

startActivity(intent);
```

Note that apps with a `targetSdkVersion` of 22 or lower are "grandfathered" into having default access to draw over other apps, simply by having requested the `SYSTEM_ALERT_WINDOW` permission. However, the user can still go into the Settings app and revoke that capability, in which case attempting to draw over another app will result in a `SecurityException`

```
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime: FATAL EXCEPTION: main
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime: Process: com.commonsware.android.debug.videolist, PID: 29661
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime: java.lang.SecurityException: com.commonsware.android.debug.videolist
from uid 10167 not allowed to perform SYSTEM_ALERT_WINDOW
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at android.os.Parcel.readException(Parcel.java:1599)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at android.os.Parcel.readException(Parcel.java:1552)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at
android.view.IWindowSession$Stub$Proxy.addToDisplay(IWindowSession.java:747)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at android.view.ViewRootImpl.setView(ViewRootImpl.java:531)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at
android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:310)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at
android.view.WindowManagerImpl.addView(WindowManagerImpl.java:85)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at
com.commonsware.android.debug.videolist.TimingWrapper.warn(TimingWrapper.java:76)
10-22 13:15:14.520 29661-29661/com.commonsware.android.debug.videolist E/
AndroidRuntime:     at
com.commonsware.android.debug.videolist.TimingWrapper.onBindViewHolder(TimingWrapper.java:55)
.
.
.
```

In many cases, there is no good way to recover from this `SecurityException`, in which case you really want to consider switching to `compileSdkVersion` of 23 or higher and calling `canDrawOverlays()` to detect this potential problem before it occurs.

Anti-Patterns

Much of this book has been focused on what you should do. In contrast, this chapter is focused on what you should *not* do.

All platforms have their anti-patterns: things that are technically possible but are not in the best interests of the users of that platform. Android is no exception. Some anti-patterns are simply annoying to users, while other anti-patterns can significantly infringe upon a user's use of their Android device, or even the user's freedom.

Much as the Hippocratic Oath directs doctors to "first, do no harm", Android application developers owe it to the users of their apps to avoid these anti-patterns to the greatest extent possible.

Prerequisites

This chapter assumes that you have read much of the book, particularly the core chapters.

Leak Threads… Or Things Attached to Threads

Leaking a thread means that you start a thread and never cause it to stop. For example, you might start a thread that runs in an infinite loop, doing some work and then sleeping for a while. The problem with infinite loops is that "infinite" is an awfully long time.

All threads should clean up, in a timely fashion, when the component (e.g., activity, service) that started the thread is destroyed — or, in the case of an activity, perhaps just moved into the background.

How you ensure that the thread gets cleaned up is up to you. For threads doing transactional work, such as literally running a database transaction, it may be fine to just let them run to completion and shut down of their own accord. For "infinite" loops, there should be some way to tell the thread that it is no longer needed, such as via an `AtomicBoolean` flag, or using something more structured than a plain timing loop, such as a `ScheduledExecutorService`.

Also, bear in mind that you are responsible for threads that are created, on your behalf, by other things that you do. The most common leak scenario here comes with listeners associated with system services, like `LocationManager` and `SensorManager`. If you register a `LocationListener` via `requestLocationUpdates()` and fail to unregister that listener, you will not only be leaking the listener, but the component associated with that listener, and every system resource tied to that listener, such as any background threads.

The Costs

Threads are intrinsically static in scope. Hence, any object they can reach, directly or indirectly, cannot be garbage-collected while the thread is still running. Hence, if an activity forks a thread, it might do so using an anonymous inner class:

```
new Thread() {
  public void run() {
    // do something
  }
}).start();
```

Instances of an inner class — anonymous or otherwise — have an implicit reference back to the object that created them. Hence, the `Thread` would hold onto the `Activity` that created the thread, which in turn would hold onto all of its widgets and so forth. None of that can be garbage-collected until after the thread terminates, even if the activity is destroyed.

The Counter-Arguments

I want the thread to keep running even after the activity is destroyed

In this case, the thread should be created and managed by a service, not simply leaked. Not only does this give you an opportunity to clean up the thread when needed, but it also alerts Android that you are still trying to do some work, so Android will not necessarily terminate your process very quickly.

However, be careful about assuming that you can have a thread — even one managed by a service — run forever, as you will see in the next couple of sections.

I do not know when the thread is no longer needed

Then you have a serious design problem.

A common variation on this theme is:

The thread is needed so long as I have an activity in the foreground

This is a bit tricky, as Android does not really expose the concept of *applications* being in the foreground, just activities.

The safest course of action is to have the thread be managed by a service, then keep track of whether or not you have an activity in the foreground. For example, in `onPause()` of each activity, use `postDelayed()` to return control to you after a short delay, and in `onResume()`, update a timestamp of your last return to the foreground (held in a static data member). When the `Runnable` for `postDelayed()` executes, check that timestamp — if it is too old, you know that none of your activities are in the foreground, and you can stop the service, having it stop your thread.

Use Large Heap Unnecessarily

Encountering an `OutOfMemoryError` certainly sucks. These are caused either by a memory leak or by trying to use more memory than is practical given the device. For example, loading up lots of bitmaps can easily chew up your available heap space.

To some, therefore, `android:largeHeap` seems to be the perfect solution.

Added in API Level 11, `android:largeHeap` tells Android to give you a much larger heap size than is normally given to a process. So, instead of having 32MB or 48MB or so of heap, you might have 256MB of heap.

The right solution, in most cases, is to fix the underlying memory problem, not to mask it by requesting an over-sized heap.

The Costs

To you, having hundreds of megabytes of extra heap may be a blessing. To the user, it may be a curse. That memory has to come from somewhere, and the "somewhere" is from other processes. Your app will force other apps' processes to be terminated far more quickly than normal, which may slow the user down when she tries to switch between your app and others. Your app may even materially harm the functionality of other apps, who have their processes terminated before they can finish their work, just to satisfy your memory craving.

Bear in mind that Android does not employ swap space (the Linux equivalent of a Windows pagefile). Hence, whereas Windows can allocate lots of memory and slows down as it goes, Android is far more limited, in accordance with its mobile roots.

Furthermore, in many cases, adding more heap space does not eliminate the problem, any more than spraying air freshener gets rid of the dead cat in your living room that is causing the odor. With a memory leak, for example, all the larger heap does is increase the time before you eventually run out of memory.

The Counter-Arguments

I really need to be able to manipulate large chunks of memory

There are certainly apps for which `android:largeHeap` is justified, such as complex data editors, such as image editors, video editors, etc.

Hence, in practice, the real anti-pattern is not using `android:largeHeap`, but rather in doing so for apps where the user would not feel that the resulting effects are justified. For example, neither a Twitter client, nor a banking app, should need a large heap, even if the developer is running into memory management issues.

Android makes it too hard to manage memory, so I need a large heap

There is no question that developing mobile applications is challenging, particularly when it comes to memory management. That is not unique to Android — embedded systems developers are used to writing apps where the heap size is better measured in KB instead of MB, for example.

Outside of bitmaps and massive data sets, though, it is a bit difficult to actually run out of memory. While a `TextView` may take up 1KB of heap space, it takes a *lot* of `TextView` widgets to chew through a 48MB heap.

The reason why bitmaps tend to trip up developers is that Android makes using them *too easy*. For example, it is simple to set a bitmap as a background of some container like a `LinearLayout`, where developers then blindly ignore the fact that if the bitmap is not *precisely* the size of the container, Android will need to scale the image, consuming more heap space.

Misuse the MENU Button

The MENU button on Android devices is designed to display either the options menu (on Android 1.x/2.x devices that are not using an action bar backport or the action bar overflow menu).

The MENU button is *not* designed for any other purpose. Some developers have taken to using it for arbitrary aims, and that is a mistake.

The Costs

The MENU button does not exist on many Android devices. In particular, devices designed for Android 3.0 and higher do not *need* a MENU button. Some will have them, but most will not. Hence, anything that requires the MENU button will simply be unavailable on those devices.

And, as of Android 4.4, Google is putting increasing pressure on device manufacturers to dump the MENU button, making it less likely to appear in the future.

The Counter-Arguments

Well, if I keep `targetSdkVersion` below 11, I can have a soft MENU button

This is true, insofar as a menu affordance will be added to the system bar or navigation bar on devices that lack a dedicated MENU button.

Whether the *user* is expecting to use this button is another thing entirely.

As more and more users run Android 3.0+ devices, they will use more and more apps that have `android:targetSdkVersion` set to 11 or higher. The remaining handful of apps that do not will be “weird”. In particular, they may not notice the menu affordance, as they are not looking for one, or they may not know what it does, as they are not used to needing it.

Moreover, eventually, other things will drive you to want an `android:targetSdkVersion` higher than 10, as the menu affordance is not the only feature driven by this value. The sooner you can remove your dependence on a menu affordance, the sooner you can upgrade your `android:targetSdkVersion` to solve other problems that you are encountering.

I think the action bar is ugly, a waste of space, or otherwise bad

That's nice. It does not mean that you need a menu affordance and a tie to a MENU button.

For example, well-written games will have a menu integrated into the game UI itself. This was often done even before Android 3.0, since the options menu UI would not look much like the game's UI, and the developer wanted a consistent look-and-feel.

So long as the user recognizes how to reach the menu (e.g., a three-dots or three-bars icon), the menu does not have to be driven by Android, but instead could be handled by your app directly. You can see this in the Google Navigation app, which avoids an action bar but still displays its own menu from its own on-screen menu affordance.

Interfere with Navigation

Some developers try to take over the device. They attempt to block the use of anything not related to their app: the HOME key, the recent tasks list, the notification drawer, etc.

Android treats such behavior as malware. Android is designed to keep control of the device in the hands of the user and tries very hard to prevent apps from stealing that control.

The Costs

While there are certain cases where blocking navigation outside the app may seem justified (see the counter-arguments, below), there is simply too much opportunity for malfeasance. Users tend to want to use their devices on their terms, not necessarily the terms of some random developer. Malware authors, in particular, love to learn about script-kiddie hacks that allow them to control a device, and by extension, control the users.

The Counter-Arguments

I am writing a lock screen

No, you are not. You are writing something that you *think* is a lock screen. Really what you are writing is something that weakens device security... if the app in question is designed to be downloaded and run on arbitrary devices.

Android devices can be rebooted into "safe mode". Much like the Windows boot option that bears the same name, "safe mode" only runs apps that are part of the system firmware, not any third-party apps.

So, let's assume that the user installs your "lock screen". Inevitably, part of the setup of a third-party "lock screen" is to disable any sort of security that is part of the native lock screen, so the user does not have to unlock things twice. Even though your lock screen may implement all sorts of security, all somebody else has to do is reboot the device in safe mode, and they now have complete access to the device, including the ability to uninstall your lock screen. By contrast, the native lock screen is in force even if the device reboots in safe mode.

I am writing a parental control app

Rebooting in safe mode is within the motor-control skills of your average three-year-old child. Hence, the primary limitation is whether or not the child knows *how* to reboot the device in safe mode, which they can learn from the Internet, friends, etc. And, if the device is really an adult's device, where the "lock screen" allows access to a subset of child-friendly apps, the real risk is not from the child rebooting the device in safe mode, but from the crook who steals the device rebooting in safe mode.

I am writing a lock screen designed to run on whole-disk-encrypted devices

While whole disk encryption — available on Android 4.0+ — does solve the issue of rebooting in safe mode, bear in mind that users then cannot disable the required password security on the native lock screen, as that is tied into the whole disk encryption process.

I am writing a kiosk app

Here, the term "kiosk app" refers to an app that represents the functionality of a single-purpose device. For example, a restaurant might want to distribute menus to customers in the form of a tablet app; the menu app would be the "kiosk app".

In this case, the owner of the device is the one trying to lock it down to be single-purpose. That is completely reasonable... except that it runs counter to the behavior of standard consumer builds of Android.

The right solution, in this case, is to create custom firmware for the single-purpose devices. This firmware can set up the kiosk app to be the home screen (thereby blunting the effectiveness of HOME, BACK, etc.), and modifications to the firmware can apply access controls to other aspects of the device (e.g., notifications). Unfortunately, there are few (if any) businesses set up to help create such single-purpose firmware for single-purpose devices.

Use android:sharedUserId

If you are creating more than one application, where those applications should be sharing data, you may be tempted to use `android:sharedUserId`. This attribute, applied to the root `<manifest>` element in your manifest, allows two or more apps to share a Linux user account. That will allow these apps full access to the other apps' files. The limitations are that you must use the same value for `sharedUserId` and that all such apps must be signed with the same signing key.

However, this is a fairly crude and somewhat risky approach to sharing information between apps. In most cases, you will be better served using any of the structured IPC options within Android, such as remote services and content providers.

The Costs

First, you must make the decision to use `android:sharedUserId` before you ever ship your app in production. Should you change the `sharedUserId` value — or switch from no value to a new value — when your change is installed, the new version of your app will have no rights to access the old version of your app's files. This is unlikely to turn out well.

Second, it will be up to you to maintain data integrity of these files in the face of simultaneous access from multiple apps. SQLite should handle this for you for your databases, as it is set up to use process-level locking — this is why SQLite can be used as the out-of-the-box database solution for Web frameworks like Rails.

However, any other sort of file, including `SharedPreferences`, will lack that coordination, unless you somehow arrange to do it yourself. And even the SQLite-level coordination has its limits, as one app has no way to know about another app's changes to the data, except by re-querying the database.

Third, using `android:sharedUserId` limits your flexibility. You cannot use it with third-party apps. You cannot readily sell one of your apps in your suite, as then it becomes a third-party app and can no longer be signed by the same signing key as are the rest of your apps. Basically, `sharedUserId` causes multiple separate APKs to behave, in some respects, as one larger APK.

The Counter-Arguments

I need to ensure only my apps can share the data, not others

Use a signature-level permission. This gives you the same level of security as does `android:sharedUserId` without most of the risks.

Writing IPC code is tedious

So is writing cross-process data integrity code.

Implement a "Quit" Button

Perhaps the most contentious question and answer on Stack Overflow's `android` tag is "Quitting an application - is that frowned upon?". This exchange is nearly three years old (as of the time of this writing), yet the answer receives both upvotes (and a few downvotes) with some regularity.

Other Android experts, such as Reto Meier, have weighed in on the issue and have offered similar recommendations – that is, do not have a "quit" or "exit" button in your app.

(here, "button" is shorthand for any command-style interface, and includes menu options, action bar items, and the like by extension)

The reason is simple: whatever your "quit" or "exit" button does *should be happening in other conditions as well*, and handling those other conditions should eliminate the need for the button.

If the app moves into the background *for any reason*, you need to treat the user and her device with respect. This means stopping background threads that are not needed, releasing system resources like the GPS radio (immediately or after a modest delay), and the like. The user should not need to "quit" your app to accomplish this, because your app will move to the background for other reasons, such as incoming phone calls, or the user pressing the HOME button.

The Costs

You might think "well, what's the harm in having the 'quit' button that, say, just calls `finish()`?"

First, rarely is it that simple. Calling `finish()` will return the user to the previous activity, and so for any multi-activity app, there will be scenarios where `finish()` is not really "quit". The only simple thing you can universally do is have "quit" bring up the home screen, in which case all you have done is waste screen real estate duplicating the HOME button functionality. Worse, the developer might say "oh, well, I will just terminate my process when they press 'quit'", and *that* anti-pattern is coming up next in this chapter.

Second, the user will start to think that they *need* to press "quit", or else bad things might happen. They will see an explicit "quit" option and start to wonder "well, gee, when am I supposed to press that, and what happens if I do not?" This, in turn, will lead to the user going out of their way to make sure to press your "quit" button, even if doing so does not actually change anything about the behavior of your app, courtesy of the placebo effect.

The Counter-Arguments

I need to let the user log out of the app, so I need a "quit" button

No, you need a "logout" button that clears your cached authentication credentials (e.g., sets a static data member to `null`), then brings up the login activity using `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to wipe out all other activities in your process. And, probably, you need to have some sort of inactivity-based "timeout" that also logs out the user (e.g., sets that static data member to `null`).

I am running stuff in the background, so I need a "quit" button

No, you need a "stop that background stuff" button, preferably with a shorter, more specific label. And, you need that to also be available from the `Notification` that you are using with your foreground service, where applicable.

Terminate Your Process

Closely related to the above anti-pattern is to forcibly stop your process, such as via `System.exit()`, `Runtime.exit()`, `Process#killProcess()`, and so forth. These are often used in concert with an in-app "quit" button, or sometimes for other reasons (e.g., could not figure out how to handle an exception gracefully).

The Costs

Simply put, Google has warned, repeatedly, that there may be side effects from terminating your own process, rather than having Android do proper cleanup first.

- "You should really think about not exiting the application. This is not how Android apps usually work." (Romain Guy)
- "To be clear: using System.exit() is strongly recommended against, and can cause some poor interactions with the system. Please don't design your app to need it." (Dianne Hackborn)
- "There is no reason or need to call [exit()]" (Dianne Hackborn)
- "Nobody has said anything about Process.kill() not doing anything. You want to kill your *own* process and cause the user to experience your *own* application having weird behavior at times due to it? Have at it. I just want to be clear that this is not what we recommend doing... and you are likely to cause bad behavior in your app at least at times due to it... There is no API to quit an application, because there is no such concept on Android, and trying to implement such a thing is going to result in fighting against how Android works." (Dianne Hackborn)

The Counter-Arguments

I am using a C library that is buggy, so I need to terminate my process

Fix the bugs in the library. For example, C libraries that rely too heavily on global variables may need to be adjusted to use session handles that get passed around.

Well, it is not my C library, but one from a third party, so I need to terminate my process

Find a library that is Android-compatible, then. It is likely that you will encounter other problems with this library, if it is not designed to work on Android (e.g., not set up to work properly on ARM CPUs).

There is a bug in Android for which I have found no workaround short of terminating my process

This is one of the few legitimate reasons for terminating a process, but it is so rare that it is difficult to find a citation of a place where such a bug (and workaround) exists.

I need to do *something* from my top-level exception handler!

Set relevant static data members to `null`, then start up your launcher activity, using `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to wipe out all other activities in your task. This should reset you to your original state, as if the user had launched the app.

Try to Hide from the User

Some developers view the user as the enemy. These developers try to insulate their app from the user, to make data inaccessible to the user, to make the app "unkillable" by the user, etc. In many cases, this is at the behest of some enterprise, wanting to exert control over the user's use of the app or even the device.

Android is a consumer operating system. It is designed to put power in the hands of whoever is holding the device and can authenticate themselves to the device (e.g., via a password on the lock screen). Enterprises and malware authors have much the same interests: they wish to take control away from the user and give the control to somebody else. Android defends against malware; enterprises get caught in the crossfire.

Inevitably, the right solution here will be an enterprise remix of Android, designed to be loaded on enterprise-supplied devices, that put the control in the hands of the enterprise.

The Costs

Simply put, you are wasting your time, which could be better spent on other pursuits.

With respect to data, if your app can access that data, by definition, a sufficiently talented user can get at the data:

- If you put it on internal storage, the user can root the device
- If you further encrypt the data, the user can find the encryption algorithm and key in your app, then decrypt the data
- If you try obfuscation or other techniques to mask the encryption algorithm and key, the user will use cracking tools to find this information anyway, or will transfer your app to a ROM mod that contains a modified version of the Android framework that can collect this information when you go to decrypt the data
- And so on

With respect to the process, the user can force-stop any installed app via the Settings app. And, even if you use script-kiddie tricks to try to prevent access to Settings, the user can nuke your app from orbit via the command line, using the full Android SDK or third-party tools.

The Counter-Arguments

I am creating an app for an enterprise, and we need to control the app

Then you further need to control the device, which leads to the "enterprise flavor of Android" solution mentioned earlier in this section.

I am creating a lock screen/parental control app/kiosk app

Please see the counter-arguments for "Interfering with Navigation" from earlier in this chapter.

Use Multiple Processes

Some Android professionals recommend the use of `android:process` to have components run in separate processes from the main one for an application. For example, you might have all of your activities in the main process but isolate a service in a separate process. Or, you might have some memory-intensive activity (e.g., an image editor) run in a separate process.

As with most of these anti-patterns, while the `android:process` feature is valid, it is rarely necessary. To some extent, developers get caught up in process isolation from

its use on servers and forget that mobile devices typically have fewer resources — RAM and CPU — than do their server counterparts. Few of Google's apps use `android:process`; even complex apps like Gmail or the original Browser avoid it.

The Costs

Each process gets its own heap space, cutting into the heap available for other applications. As with the large-heap anti-pattern discussed above, this will tend to force other apps to be ejected from memory sooner than normal, with commensurate impacts on user experience.

Inter-process communication (IPC) is not cheap, compared with normal method invocation within a process. Hence, tightly-coupled processes will chew through more CPU than their single-process counterparts. While it is unlikely that you will see major performance implications (unless you are doing a preposterous amount of IPC), this will consume more battery than is otherwise warranted.

The Counter-Arguments

I am using a C library that is buggy, and you told me not to terminate my process

As noted earlier, fix the bugs in the library.

Hello? It is not my C library, but one from a third party!

Find a library that is Android-compatible, then.

I need more heap space

On Android 3.0 and higher, `android:largeHeap` is available, though its misuse is another anti-pattern, discussed above. However, prior to Android 3.0, `android:largeHeap` was not an option. One workaround used by some apps is to fork several processes, thereby getting several "small" heap allocations (e.g., 32MB) instead of just one.

In cases where `android:largeHeap` is indeed justified, using multiple processes as a workaround on older Android versions is justified as well. However, bear in mind that IPC overhead is non-trivial, so have a plan to dump the multiple processes and use `android:largeHeap` once you drop support for Android 1.x/2.x.

I want my UI not to freeze when doing background work

Use threads, not processes, for this.

Hog System Resources

Some of these anti-patterns, like the multiple-process one just now, are really concrete sub-types of a more general anti-pattern: assuming yours is the only app running on the device. While your app may be the only one running in the foreground (assuming that you actually *are* in the foreground), there are other apps in the background, and ones that soon will come to the foreground. You need to "play nice" and ensure that these other apps will have their fair share of system resources.

One example is open files on external storage. For some devices — but not all – there is a limit of 1,024 simultaneously open files. In principle, that should be plenty. However, if some app — maybe yours? — opens a whole bunch of files, it is possible that other apps trying to access external storage at that point will crash because the limit was hit.

The Counter-Arguments

Um, well, I'm just more important than those other developers
:facepalm::

Trail: Widget Catalog

Widget Catalog: AdapterViewFlipper

A regular `ViewFlipper` shows only one child widget or container at a time. So does an `AdapterViewFlipper`. The difference is where the children come from. With a regular `ViewFlipper`, you add children much like you would any other standard container class, such as defining the children in your layout XML resource. With `AdapterViewFlipper`, the children come from an `Adapter`.

While `AdapterViewFlipper` does not inherit from `ViewFlipper` (or vice versa, for that matter), their public API is largely the same:

- You can control which child is visible, either by index or via `showNext()`/`showPrevious()` methods to rotate between them.
- You can set up animated effects to control how a child leaves and the next one enters, such as applying a sliding effect.
- You can set up `AdapterViewFlipper` to automatically flip between children on a specified period.

There are two key advantages for `AdapterViewFliper`:

1. Since it uses an `Adapter` model, it can be more memory efficient for lots of children, through child view recycling
2. It is available for use in an app widget

However, `AdapterViewFlipper` is new to API Level 11 and is unavailable on older versions of Android. It is not included in the Android Support package backport.

Key Usage Tips

All of the usage tips from `ViewFlipper` are relevant for `AdapterViewFlipper`.

A Sample Usage

The sample project can be found in WidgetCatalog/AdapterViewFlipper.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<AdapterViewFlipper xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/details"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

(from WidgetCatalog/AdapterViewFlipper/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.avflip;

import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterViewFlipper;
import android.widget.ArrayAdapter;

public class FlipperDemo2 extends Activity {
  static String[] items= { "lorem", "ipsum", "dolor", "sit", "amet",
      "consectetuer", "adipiscing", "elit", "morbi", "vel", "ligula",
      "vitae", "arcu", "aliquet", "mollis", "etiam", "vel", "erat",
      "placerat", "ante", "porttitor", "sodales", "pellentesque",
      "augue", "purus" };
  AdapterViewFlipper flipper;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    flipper=(AdapterViewFlipper)findViewById(R.id.details);
    flipper.setAdapter(new ArrayAdapter<String>(this, R.layout.big_button, items));
    flipper.setFlipInterval(2000);
    flipper.startFlipping();
  }
}
```

(from WidgetCatalog/AdapterViewFlipper/app/src/main/java/com/commonsware/android/avflip/FlipperDemo2.java)

Visual Representation

There is no visual representation of an `AdapterViewFlipper` itself, as it renders no pixels on its own. Rather, it simply shows the current child.

Widget Catalog: CalendarView

`CalendarView`, as you might have guessed, displays a calendar to the user, designed to allow the user to pick a date. You supply a starting date, which the user then manipulates, triggering event listeners whenever the date is changed.

Note that this is a small calendar – it is not designed to show details *within* a date, such as appointments and times.

This view is available standalone and also as an optional adjunct to the `DatePicker` widget.

This view was added in API Level 11 and therefore will not be available on older versions of Android, though a backport is available that works on Android 2.2 onwards.

Key Usage Tips

If you do nothing, the `CalendarView` will start with today's date, though you can call a `setDate()` method to pass in a `Calendar` object to use to change the initially-selected date. You can also call `setOnDateChangeListener()` to supply an `OnDateChangeListener` to learn when the user changes the date in the `CalendarView`.

`CalendarView` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the year/month/day-of-month from the `CalendarView` (as supplied to the `onSelectedDayChange()` method of your `OnDateChangeListener`) and converting it into something you can use in your code.

A Sample Usage

The sample project can be found in WidgetCatalog/CalendarView.

Layout:

```
<CalendarView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/calendar"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

(from WidgetCatalog/CalendarView/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.wc.calendar;

import android.app.Activity;
import android.os.Bundle;
import android.widget.CalendarView;
import android.widget.CalendarView.OnDateChangeListener;
import android.widget.Toast;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class CalendarDemoActivity extends Activity implements
    OnDateChangeListener {
  CalendarView calendar=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    calendar=(CalendarView)findViewById(R.id.calendar);
    calendar.setOnDateChangeListener(this);
  }

  @Override
  public void onSelectedDayChange(CalendarView view, int year,
                                  int monthOfYear, int dayOfMonth) {
    Calendar then=new GregorianCalendar(year, monthOfYear, dayOfMonth);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_LONG)
         .show();
```

```
  }
}
```

(from WidgetCatalog/CalendarView/app/src/main/java/com/commonsware/android/wc/calendar/CalendarDemoActivity.java)

Visual Representation

This is what a `CalendarView` looks like in a few different Android versions and configurations, based upon the sample app shown above.

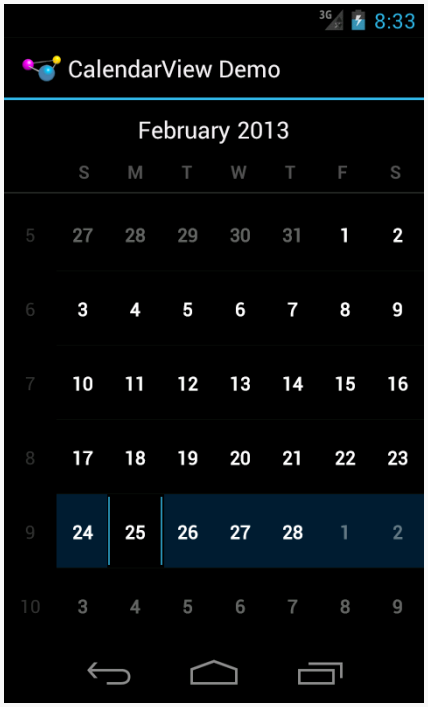


Figure 1073: Android 4.0

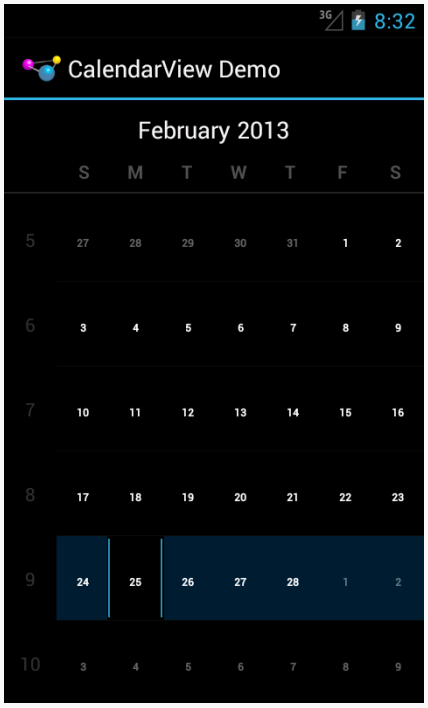


Figure 1074: Android 4.1

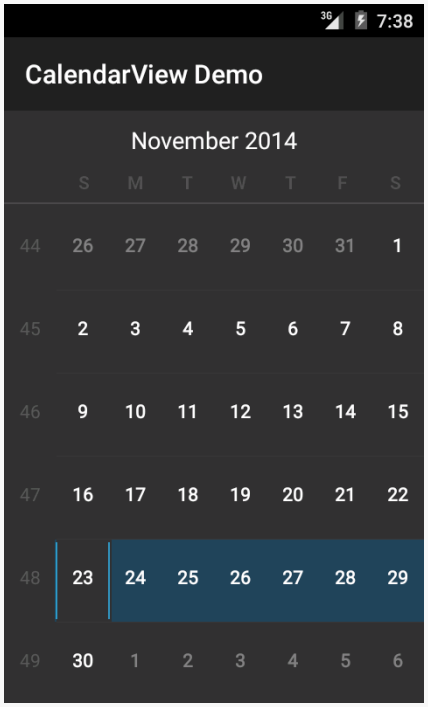
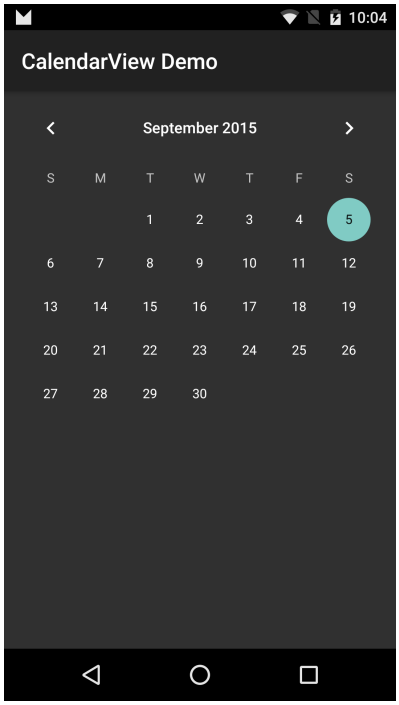


Figure 1075: Android 5.0


10:04
CalendarView Demo
September 2015
S M T W T F S
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

Widget Catalog: DatePicker

`DatePicker`, as the name might suggest, allows the user to pick a date. You supply a starting date, which the user then manipulates, triggering event listeners whenever the date is changed.

Key Usage Tips

If you do nothing, the `DatePicker` will start with today's date. However, if you want to set up an `OnDateSetListener` to find out when the date changes, you will need to call `init()` to do so, in which you also need to set the date.

`DatePicker` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the year/month/day-of-month from the `DatePicker` and converting it into something you can use in your code.

API Level 11 introduced an optional `CalendarView` adjunct to the `DatePicker`, determined via `setCalendarViewShown()` or `android:calendarViewShown`. This works well on `-normal` screens in landscape and on `-large`/`-xlarge` screens. On `-normal` screens in portrait, the year portion of the picker may be chopped off to save room. Using the `CalendarView` option on `-small` screens is probably not a good idea.

However, on Android 5.0+, the `CalendarView` is *always* shown and cannot be removed, as the "picker" itself does not allow the user to pick a date. The user uses the `CalendarView` to pick a date, or taps on the year in the "picker" to choose a year. This means that `DatePicker` is not a particularly good widget to use, especially on smaller screens.

A Sample Usage

The sample project can be found in WidgetCatalog/DatePicker.

Layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center_horizontal">

  <DatePicker
    android:id="@+id/picker"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:calendarViewShown="true"/>

  <CheckBox
    android:id="@+id/showCalendar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:checked="true"
    android:text="@string/calendar"/>

</LinearLayout>
```

(from WidgetCatalog/DatePicker/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.wc.datepick;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;
import android.widget.DatePicker;
import android.widget.DatePicker.OnDateChangedListener;
import android.widget.Toast;
import java.util.Calendar;
import java.util.GregorianCalendar;
```

```
public class DatePickerDemoActivity extends Activity implements
    OnCheckedChangeListener, OnDateChangedListener {
  DatePicker picker=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    CheckBox cb=(CheckBox)findViewById(R.id.showCalendar);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      cb.setOnCheckedChangeListener(this);
    }
    else {
      cb.setVisibility(View.GONE);
    }

    GregorianCalendar now=new GregorianCalendar();

    picker=(DatePicker)findViewById(R.id.picker);
    picker.init(now.get(Calendar.YEAR), now.get(Calendar.MONTH),
                now.get(Calendar.DAY_OF_MONTH), this);
  }

  @Override
  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      picker.setCalendarViewShown(isChecked);
    }
  }

  @Override
  public void onDateChanged(DatePicker view, int year, int monthOfYear,
                            int dayOfMonth) {
    Calendar then=new GregorianCalendar(year, monthOfYear, dayOfMonth);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_LONG)
         .show();
  }
}
```

(from WidgetCatalog/DatePicker/app/src/main/java/com/commonsware/android/wc/datepick/DatePickerDemoActivity.java)

The `CheckBox` is tied to the visibility of the `CalendarView`. Since this is only available on API Level 11 and higher, we simply remove the `CheckBox` on earlier versions of

Android, so we do not have to worry about whether or not the `CheckBox` gets unchecked by the user.

Visual Representation

This is what a `DatePicker` looks like in a few different Android versions and configurations, based upon the sample app shown above.

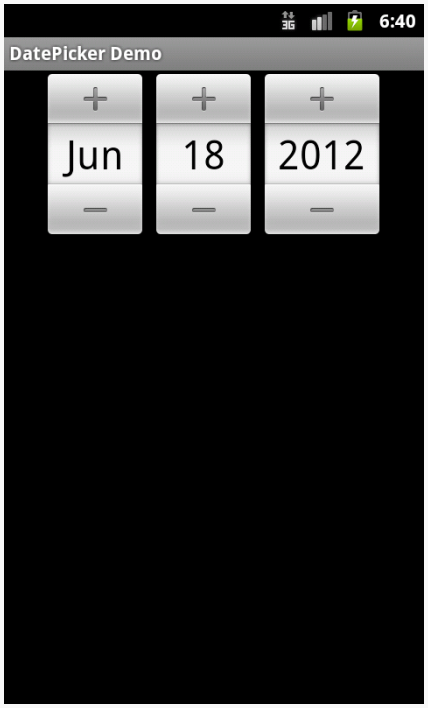


Figure 1076: Android 2.3.3

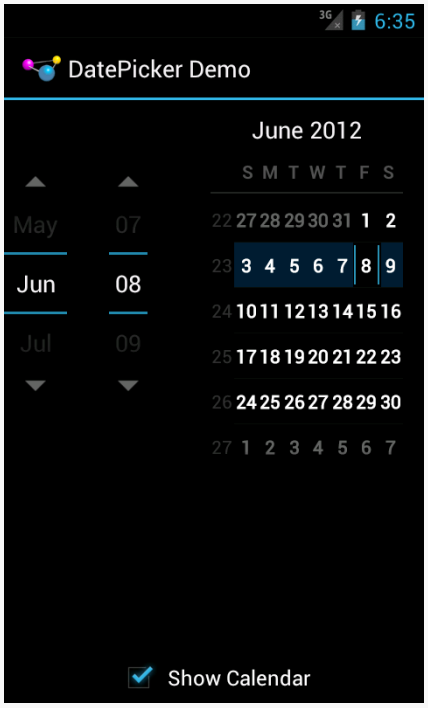


Figure 1077: Android 4.0.3, with CalendarView, Portrait

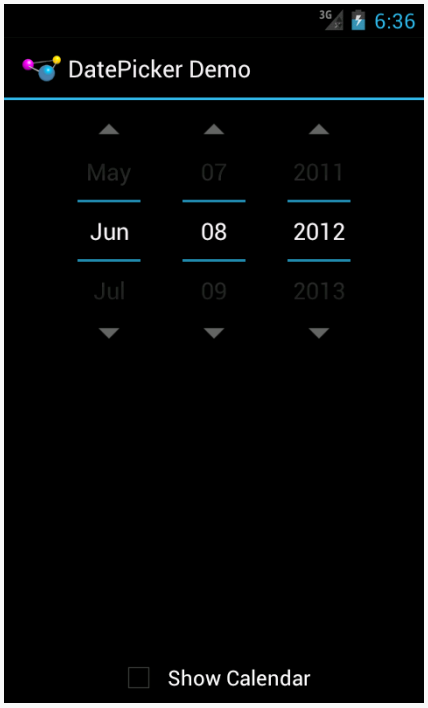


Figure 1078: Android 4.0.3, without CalendarView, Portrait

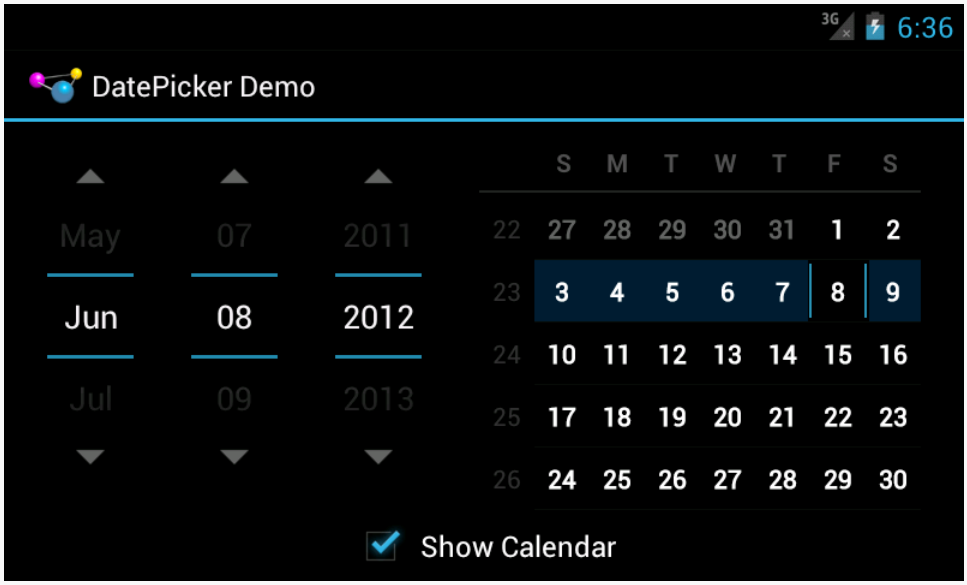


Figure 1079: Android 4.0.3, with CalendarView, Landscape

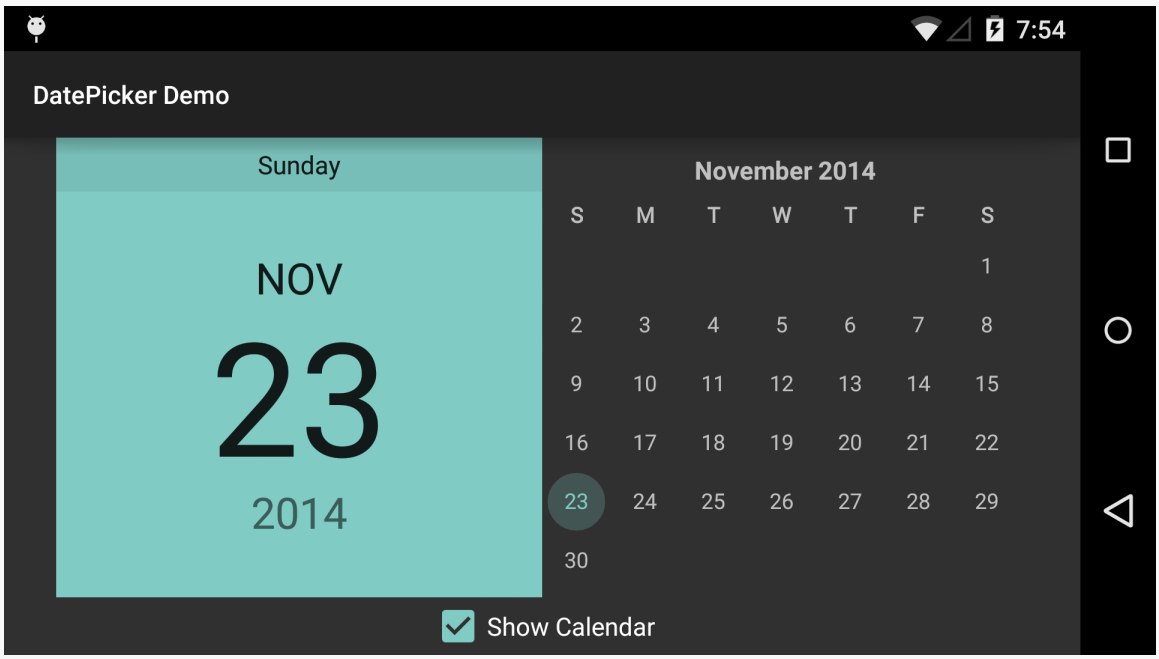


Figure 1080: Android 5.0, with CalendarView, Landscape

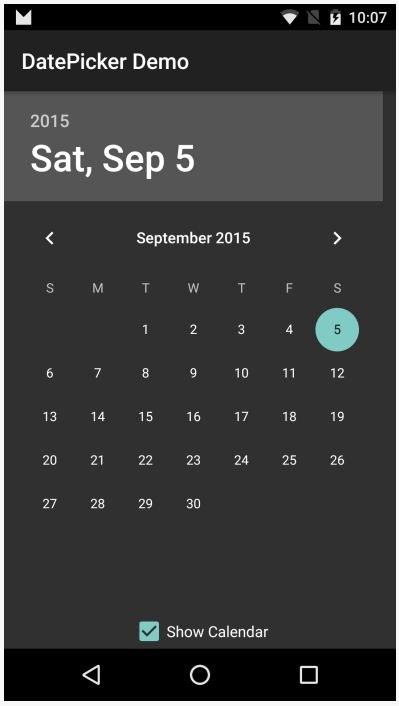


Figure 1081: Android 6.0, with CalendarView, Portrait

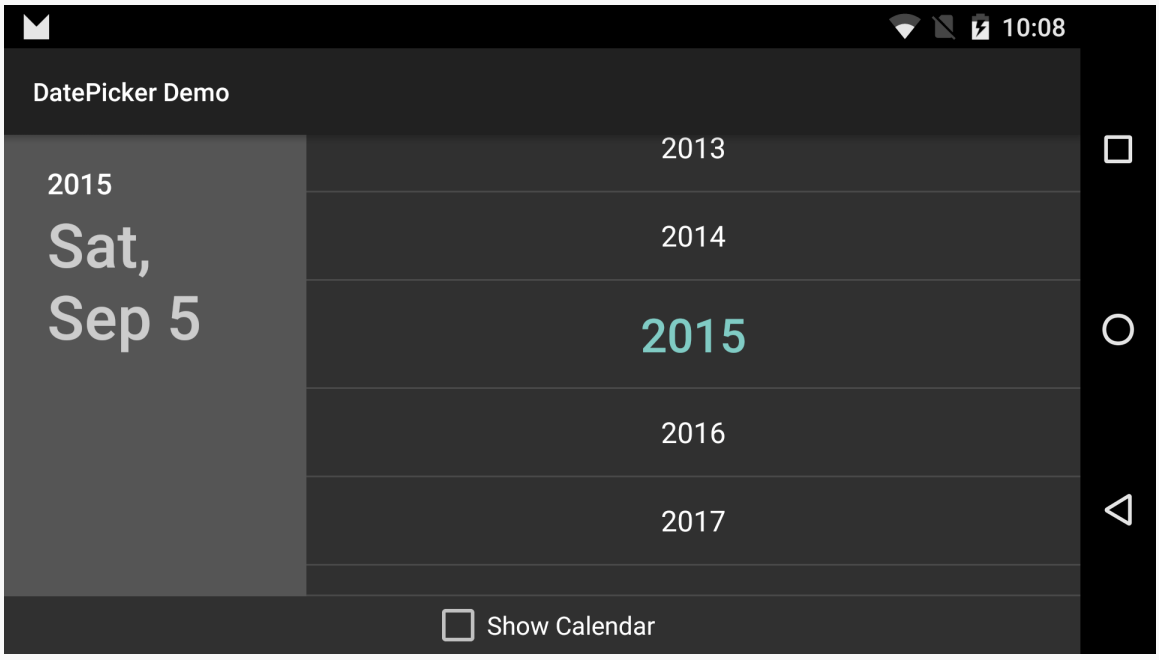


Figure 1082: Android 6.0, Showing Year Picker, Landscape

—

Widget Catalog: ExpandableListView

Android does not have a "tree" widget, allowing users to navigate an arbitrary hierarchy of stuff. In large part, that is because such trees are difficult to navigate on small touchscreens with comparatively large fingers.

Android *does* have `ExpandableListView`, a subclass of `ListView` that supports a two-layer hierarchy: groups and children. Groups can be expanded to show their children or collapsed to hide them, and you can get control on various events for the groups or the children.

Key Usage Tips

Android offers an `ExpandableListActivity` as a counterpart to its `ListActivity`. However, it does not offer an `ExpandableListFragment`. This is not a major issue, as you can work with an `ExpandableListView` inside a regular `Fragment` yourself, just as you would for most other widgets not named `ListView`.

Rather than use a `ListAdapter` with `ExpandableListView`, you will use an `ExpandableListAdapter`, where you can control separate details for groups and children. These include:

- `SimpleExpandableListAdapter`, roughly analogous to `ArrayAdapter`, where your data resides in a `List` of `Map` objects for groups, and a `List` of a `List` of `Map` objects for the children
- `CursorTreeAdapter` and `SimpleCursorTreeAdapter`, roughly analogous to `CursorAdapter` and `SimpleCursorAdapter`, for mapping data in a `Cursor` to rows and columns

In many cases, though, the complexity of managing groups and children will steer you down the path of extending `BaseExpandableListAdapter` and handling all of the view construction yourself. There are many methods that you will need to implement:

- `getGroupCount()`, to return the number of groups
- `getGroup()` and `getGroupId()`, to return an `Object` and unique `int` ID for a group given its position
- `getGroupView()`, to return the `View` that should be used to render the group, perhaps using the built-in `android.R.layout.simple_expandable_list_item_1` that is set up for such groups and handles rendering the expanded and collapsed states
- `getChildrenCount()`, to return the number of children for a given group
- `getChild()` and `getChildId()`, to return an `Object` and unique `int` ID for a child given its position (and its group's position)
- `getChildView()`, to return the `View` that should be used to render the child, given its position and its group's position
- `isChildSelectable()`, to indicate if the user can select a given child, given its position and its group's position
- `hasStableIds()`, to indicate if the ID values you returned from `getGroupId()` and `getChildId()` will remain constant for the life of this adapter

There are four major events that you will be able to respond to with respect to the user's interaction with an `ExpandableListView`:

- Clicks on a child (`setOnChildClickListener()`)
- Clicks on a group (`setOnGroupClickListener()`)
- When groups expand (`setOnGroupExpandListener()`) or collapse (`setOnGroupCollapseListener()`)

If you use `setOnGroupClickListener()` to be notified about clicks on a group, be sure to return `false` from your implementation of the `onGroupClick()` method required by the `OnGroupClickListener` interface. If you return `true`, you consume the click event, which prevents `ExpandableListView` from using that event to expand or collapse the group.

A Sample Usage

The sample project can be found in WidgetCatalog/ExpandableListView.

Layout:

```
<ExpandableListView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/elv"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

</ExpandableListView>
```

(from WidgetCatalog/ExpandableListView/app/src/main/res/layout/activity_main.xml)

JSON data:

```
{
  "Group A": ["Child A1", "Child A2", "Child A3"],
  "Group B": ["Child B1", "Child B2"],
  "Group C": ["Child C1"],
  "Group D": [],
  "Group E": ["Child E1", "Child E2", "Child E3"]
}
```

Activity:

```
package com.commonsware.android.wc.elv;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ExpandableListAdapter;
import android.widget.ExpandableListView;
import android.widget.ExpandableListView.OnChildClickListener;
import android.widget.ExpandableListView.OnGroupClickListener;
import android.widget.ExpandableListView.OnGroupCollapseListener;
import android.widget.ExpandableListView.OnGroupExpandListener;
import android.widget.Toast;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import org.json.JSONObject;

public class MainActivity extends Activity implements
    OnChildClickListener, OnGroupClickListener, OnGroupExpandListener,
    OnGroupCollapseListener {
  private ExpandableListAdapter adapter=null;

  @Override
```

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  InputStream raw=getResources().openRawResource(R.raw.sample);
  BufferedReader in=new BufferedReader(new InputStreamReader(raw));
  String str;
  StringBuffer buf=new StringBuffer();

  try {
    while ((str=in.readLine()) != null) {
      buf.append(str);
      buf.append('\n');
    }

    in.close();

    JSONObject model=new JSONObject(buf.toString());

    ExpandableListView elv=(ExpandableListView)findViewById(R.id.elv);

    adapter=new JSONExpandableListAdapter(getLayoutInflater(), model);
    elv.setAdapter(adapter);

    elv.setOnChildClickListener(this);
    elv.setOnGroupClickListener(this);
    elv.setOnGroupExpandListener(this);
    elv.setOnGroupCollapseListener(this);
  }
  catch (Exception e) {
    Log.e(getClass().getName(), "Exception reading JSON", e);
  }
}

@Override
public boolean onChildClick(ExpandableListView parent, View v,
                            int groupPosition, int childPosition,
                            long id) {
  Toast.makeText(this,
                 adapter.getChild(groupPosition, childPosition)
                        .toString(), Toast.LENGTH_SHORT).show();

  return(false);
}

@Override
public boolean onGroupClick(ExpandableListView parent, View v,
                            int groupPosition, long id) {
```

```
    Toast.makeText(this, adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();

    return(false);
  }

  @Override
  public void onGroupExpand(int groupPosition) {
    Toast.makeText(this,
                   "Expanding: "
                       + adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();
  }

  @Override
  public void onGroupCollapse(int groupPosition) {
    Toast.makeText(this,
                   "Collapsing: "
                       + adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();
  }
}
```

(from WidgetCatalog/ExpandableListView/app/src/main/java/com/commonsware/android/wc/elv/MainActivity.java)

This activity loads up a JSON file from a raw resource on the main application thread in `onCreate()`, which is not a good idea. It would be better to do that work in a background thread, perhaps an `AsyncTask` managed by a retained fragment. The implementation shown here is designed to keep the sample small, not to demonstrate the best way to load data from a raw resource.

Adapter:

```
package com.commonsware.android.wc.elv;

import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseExpandableListAdapter;
import android.widget.TextView;
import java.util.Iterator;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class JSONExpandableListAdapter extends
```

```
  BaseExpandableListAdapter {
LayoutInflater inflater=null;
JSONObject model=null;

JSONExpandableListAdapter(LayoutInflater inflater, JSONObject model) {
  this.inflater=inflater;
  this.model=model;
}

@Override
public int getGroupCount() {
  return(model.length());
}

@Override
public Object getGroup(int groupPosition) {
  @SuppressWarnings("rawtypes")
  Iterator i=model.keys();

  while (groupPosition > 0) {
    i.next();
    groupPosition--;
  }

  return(i.next());
}

@Override
public long getGroupId(int groupPosition) {
  return(groupPosition);
}

@Override
public View getGroupView(int groupPosition, boolean isExpanded,
                         View convertView, ViewGroup parent) {
  if (convertView == null) {
    convertView=
        inflater.inflate(android.R.layout.simple_expandable_list_item_1,
                         parent, false);
  }

  TextView tv=
      ((TextView)convertView.findViewById(android.R.id.text1));
  tv.setText(getGroup(groupPosition).toString());

  return(convertView);
}
```

```
@Override
public int getChildrenCount(int groupPosition) {
  try {
    JSONArray children=getChildren(groupPosition);

    return(children.length());
  }
  catch (JSONException e) {
    // JSONArray is really annoying
    Log.e(getClass().getSimpleName(), "Exception getting children", e);
  }

  return(0);
}

@Override
public Object getChild(int groupPosition, int childPosition) {
  try {
    JSONArray children=getChildren(groupPosition);

    return(children.get(childPosition));
  }
  catch (JSONException e) {
    // JSONArray is really annoying
    Log.e(getClass().getSimpleName(),
          "Exception getting item from JSON array", e);
  }

  return(null);
}

@Override
public long getChildId(int groupPosition, int childPosition) {
  return(groupPosition * 1024 + childPosition);
}

@Override
public View getChildView(int groupPosition, int childPosition,
                         boolean isLastChild, View convertView,
                         ViewGroup parent) {
  if (convertView == null) {
    convertView=
        inflater.inflate(android.R.layout.simple_list_item_1, parent,
                         false);
  }

  TextView tv=(TextView)convertView;
  tv.setText(getChild(groupPosition, childPosition).toString());
```

```
    return(convertView);
  }

  @Override
  public boolean isChildSelectable(int groupPosition, int childPosition) {
    return(true);
  }

  @Override
  public boolean hasStableIds() {
    return(true);
  }

  private JSONArray getChildren(int groupPosition) throws JSONException {
    String key=getGroup(groupPosition).toString();

    return(model.getJSONArray(key));
  }
}
```

(from WidgetCatalog/ExpandableListView/app/src/main/java/com/commonsware/android/wc/elv/JSONExpandableListAdapter.java)

This adapter wraps a `JSONObject` and assumes that the JSON structure is an object, keyed by strings, whose values are arrays of strings. The object returned by `getGroup()` is the key for that group's position; the object returned by `getChild()` is the string at that child's array index for it's group's array. Since the data structure is treated as immutable, and since there are no other better IDs in the data structure itself, the group ID is simply the group's position, and the child's ID is simply a mash-up of the group and child positions.

Visual Representation

This is what an `ExpandableListView` looks like in a few different Android versions and configurations, based upon the sample app shown above.

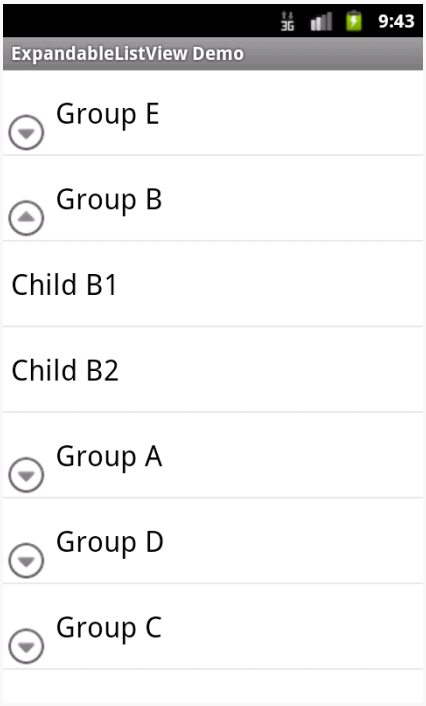


Figure 1083: Android 2.3.3, Portrait

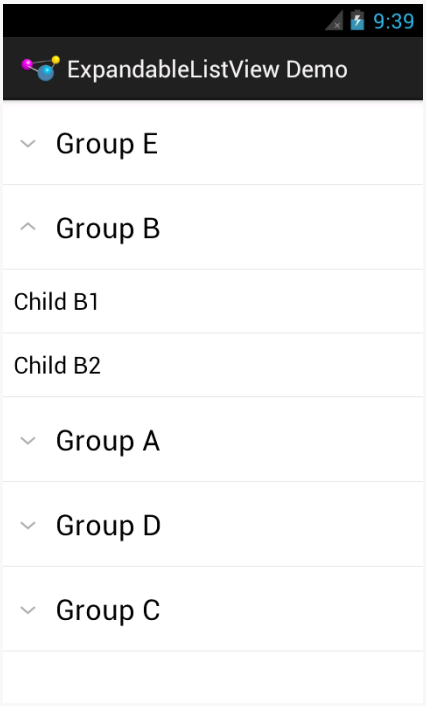


Figure 1084: Android 4.0.3, Portrait

Note that while the data in the JSON file has the groups sorted alphabetically, because `JSONObject` effectively loads its data into a `HashMap`, the sorting gets lost in the data model, which is why the groups appear out of order.

Also note that the visual representation of the "collapsed" and "expanded" states is controlled by the `ExpandableListAdapter` and the view used for the groups. In this sample, we use `android.R.layout.simple_expandable_list_item_1` for the groups, which gives us the caret designation for expanded versus collapsed states in 4.0.3 and the lower-left arrowhead-in-circle icon for 2.3.3. You can create your own rows with your own indicators as you see fit.

Widget Catalog: SeekBar

`SeekBar` allows the user to choose a value along a continuous range by sliding a "thumb" along a horizontal line. In effect — and in practice, as it turns out – `SeekBar` is a user-modifiable `ProgressBar`.

Key Usage Tips

The value range of a `SeekBar` runs from 0 to a developer-set maximum value. As with `ProgressBar`, the default maximum is 100, but that can be changed via an `android:max` attribute or the `setMax()` method. The minimum value is always 0, so if you want a range starting elsewhere, just add your starting value to the actual value (obtained via `getProgress()`) to slide the range as desired.

You can find out about changes in the `SeekBar` value by attaching an `OnSeekBarChangeListener` implementation. The primary method on that interface is `onProgressChanged()`, where you are notified about changes in the progress value (second parameter) and whether that change was initiated directly by the user interacting with the widget (third parameter). The interface also has `onStartTrackingTouch()` and `onStopTrackingTouch()`, to indicate when the user is attempting to change the position of the thumb via the touchscreen, though these methods are less-commonly used.

A Sample Usage

The sample project can be found in `WidgetCatalog/SeekBar`.

Layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center_vertical"
  tools:context=".MainActivity">

  <TextView
    android:id="@+id/value"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="0"
    android:ems="2"
    android:gravity="right|center_vertical"
    android:layout_marginRight="10dp"
    android:textAppearance="@android:style/TextAppearance.Large"/>

  <SeekBar
    android:id="@+id/seek_bar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:layout_marginRight="10dp"
    android:max="50"/>

</LinearLayout>
```

(from WidgetCatalog/SeekBar/app/src/main/res/layout/activity_main.xml)

Activity:

```
package com.commonsware.android.wc.seekbar;

import android.app.Activity;
import android.os.Bundle;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;
import android.widget.TextView;

public class MainActivity extends Activity implements
    OnSeekBarChangeListener {
  TextView value=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
```

```
    value=(TextView)findViewById(R.id.value);

    SeekBar seekBar=(SeekBar)findViewById(R.id.seek_bar);

    seekBar.setOnSeekBarChangeListener(this);
  }

  @Override
  public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
    value.setText(String.valueOf(progress));
  }

  @Override
  public void onStartTrackingTouch(SeekBar seekBar) {
    // no-op
  }

  @Override
  public void onStopTrackingTouch(SeekBar seekBar) {
    // no-op
  }
}
```

(from WidgetCatalog/SeekBar/app/src/main/java/com/commonsware/android/wc/seekbar/MainActivity.java)

Visual Representation

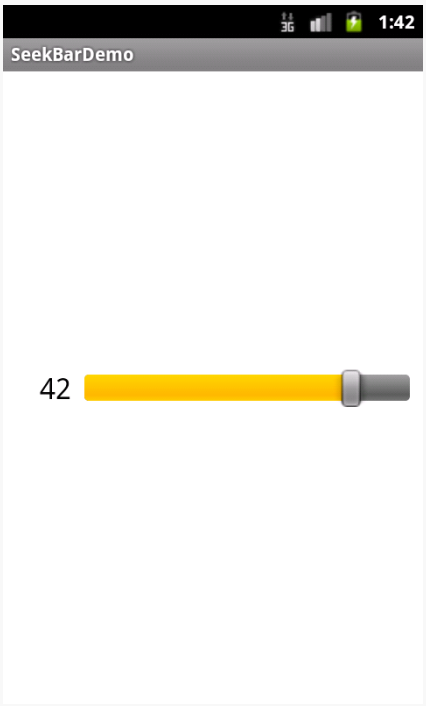


Figure 1085: Android 2.3.3

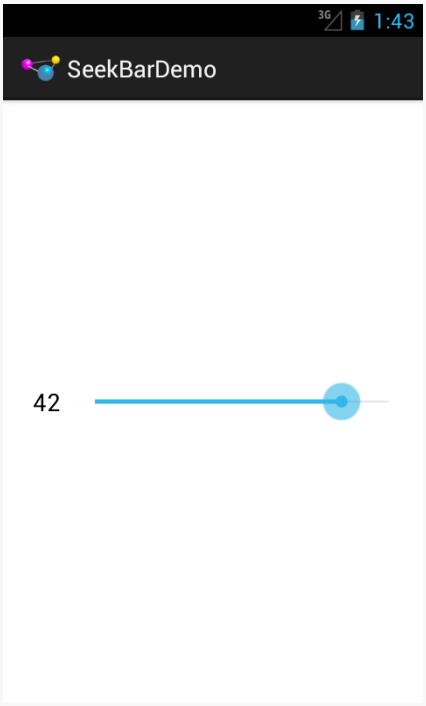


Figure 1086: Android 4.1

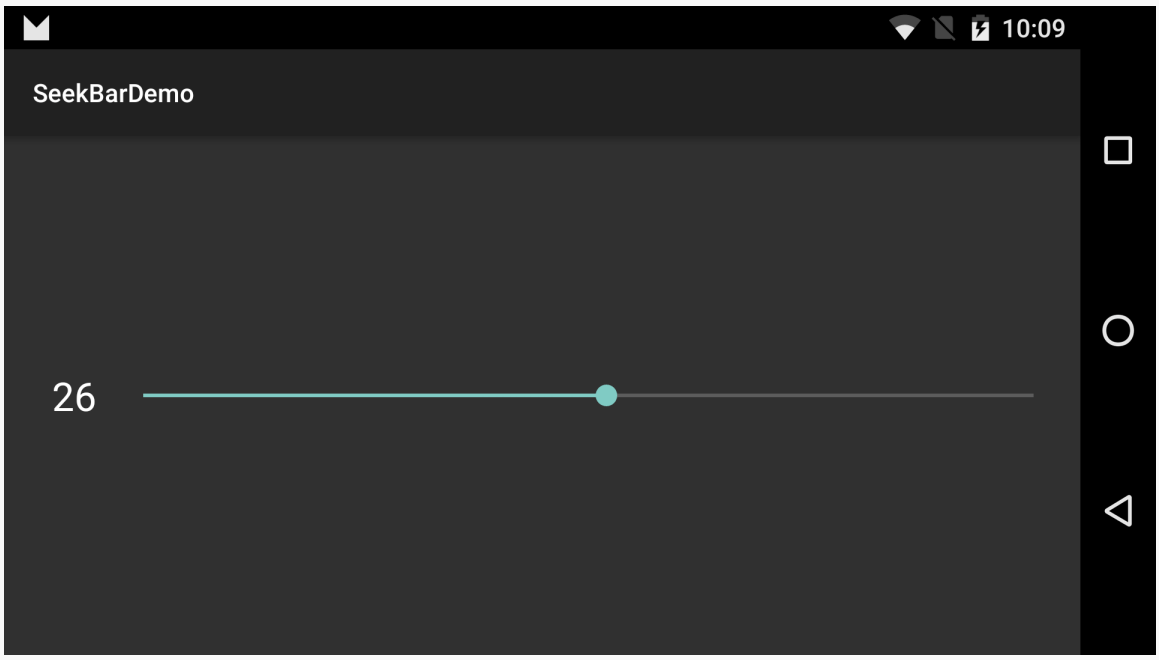


Figure 1087: Android 6.0, Landscape

Widget Catalog: SlidingDrawer

Having some form of means of allowing the user to swipe to show more things is an important visual pattern. We saw this earlier in the book with the `ViewPager` container. And there are other modern techniques for doing this that you will see in apps like Google+.

`SlidingDrawer`, while implementing a variation on this pattern, is a bit out of date at present. Mostly, that's a question of its UI: tapping a drawer "handle" to open it is not what you tend to see nowadays. That being said, it works perfectly well, wrapping around a container to make it appear or disappear based on user input, complete with a sliding animation effect.

Note that `SlidingDrawer` was deprecated in API Level 17 (a.k.a., Android 4.2). This means that Google is steering you in other directions, including forking the AOSP code for `SlidingDrawer` and maintaining it yourself. The animator framework offers other ways of implementing sliding widgets that may be better suited for your UI, anyway.

Also note that `SlidingDrawer` is broken on Android 5.0, and so you *definitely* should be considering alternative widgets at this time.

Key Usage Tips

The `SlidingDrawer` itself is transparent, except for the button to trigger the slide and its accompanying horizontal bar. Hence, if you want the drawer contents to completely obscure what is outside of the drawer, you will need to use an appropriate background. Otherwise, the drawer contents and what lies outside the drawer will be alpha-blended based on their own translucency, as is seen in the screenshots later in this chapter.

The `SlidingDrawer` can be horizontal or vertical; it is vertical by default. However, it only slides one way (bottom-to-top for vertical, right-to-left for horizontal). There is no way to reverse the direction of the sliding effect.

You must supply `android:content` and `android:handle` attributes in `SlidingDrawer`, containing references to the widget that forms the content of the drawer and the drawer's handle, respectively. Typically, the drawer's handle is an `ImageView`. Note that you *must* supply a handle — you cannot skip either of these attributes.

A Sample Usage

The sample project can be found in WidgetCatalog/SlidingDrawer.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/drawer_closed"/>

  <SlidingDrawer
    android:id="@+id/drawer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:content="@+id/content"
    android:handle="@+id/handle">

    <ImageView
      android:id="@id/handle"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/tray_handle_normal"/>

    <Button
      android:id="@id/content"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="@string/drawer_msg"/>
```

```
  </SlidingDrawer>

</RelativeLayout>
```

(from WidgetCatalog/SlidingDrawer/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.drawer;

import android.app.Activity;
import android.os.Bundle;

public class DrawerDemo extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

(from WidgetCatalog/SlidingDrawer/app/src/main/java/com/commonsware/android/drawer/DrawerDemo.java)

Visual Representation

This is what a `SlidingDrawer` looks like in a few different Android versions and configurations, based upon the sample app shown above.

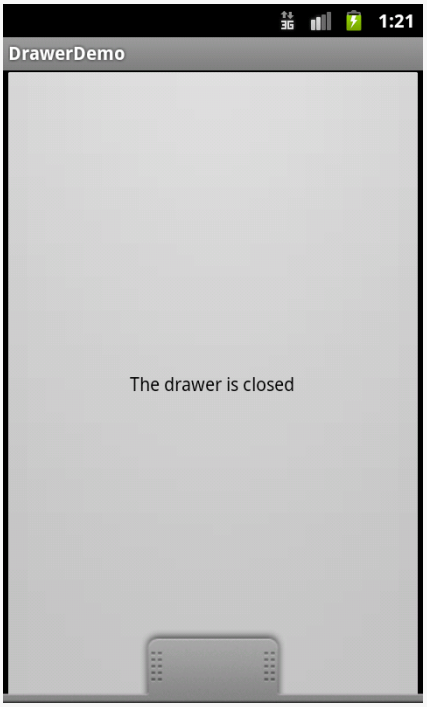


Figure 1088: Android 2.3.3, with Drawer Closed

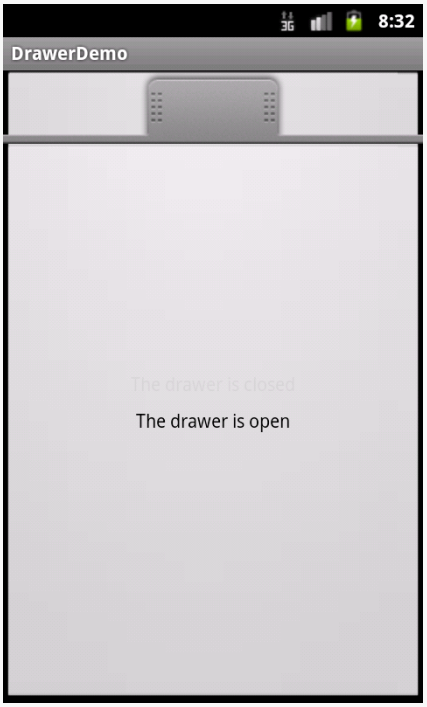


Figure 1089: Android 2.3.3, with Drawer Open

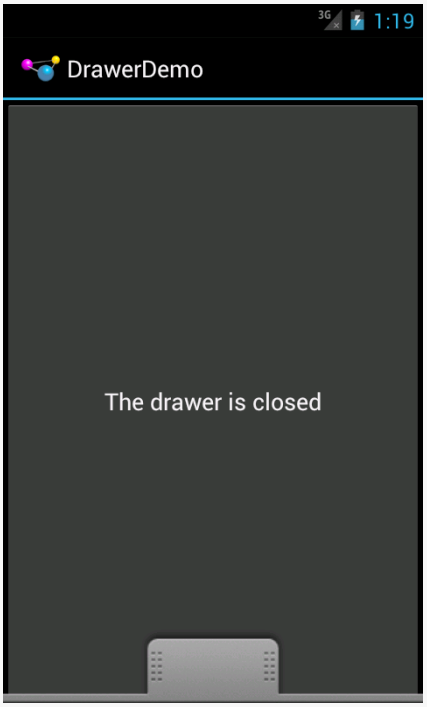


Figure 1090: Android 4.0.3, with Drawer Closed

Widget Catalog: StackView

`StackView` is an `AdapterView`. Whereas `ListView` uses a horizontal scrolling list as its UI metaphor, `StackView` uses a stack of cards as its metaphor. Just as `ListView` shows a handful of rows, `StackView` shows a handful of cards. These cards can be swiped away via a swipe towards the southwest corner of the screen. The top card is fully visible; the edges of a few other cards can be seen but are otherwise obscured by cards "higher in the stack".

While certainly usable in activities and fragments, `StackView` was introduced in support of app widgets. App widgets like bookmarks, Google Books covers, and the like use `StackView` to show an item and allow users to navigate to the rest of the items by flipping these virtual cards.

Key Usage Tips

Generally speaking, working with `StackView` is not significantly different than is working with any other `AdapterView`. You create an `Adapter` defining the contents (in this case, defining the cards), you attach the `Adapter` to the `StackView`, and put the `StackView` somewhere on the screen.

As the cards overlap, however, transparency becomes an issue. If the top card is not completely opaque, you will see the card beneath it "peeking through" as its contents are blended in via the alpha channel. In some cases, this is a perfectly desirable outcome. However, if that is not what you want, make sure that the backgrounds of your overall container for the card's contents (e.g., a `RelativeLayout`) has an opaque background, such as a color with `FF` for the alpha value.

Also, since the objective is to have the children be visually stacked, the children cannot be the size of the `StackView` itself (e.g., the children cannot use `match_parent` for a dimension). `StackView` seems to work best with children that have explicit sizes (e.g., values in `dp`).

A Sample Usage

The sample project can be found in WidgetCatalog/StackView.

Activity Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<StackView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/details"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

(from WidgetCatalog/StackView/app/src/main/res/layout/main.xml)

Item Layout:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="200dp"
  android:layout_height="200dp"
  android:background="#FFFF0000"
  android:gravity="center"
  android:textAppearance="?android:attr/textAppearanceLarge"/>
```

(from WidgetCatalog/StackView/app/src/main/res/layout/item.xml)

Activity:

```
package com.commonsware.android.wc.stack;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.StackView;

public class MainActivity extends Activity {
  static String[] items= { "lorem", "ipsum", "dolor", "sit", "amet",
      "consectetuer", "adipiscing", "elit", "morbi", "vel", "ligula",
```

```
      "vitae", "arcu", "aliquet", "mollis", "etiam", "vel", "erat",
      "placerat", "ante", "porttitor", "sodales", "pellentesque",
      "augue", "purus" };
  StackView stack;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    stack=(StackView)findViewById(R.id.details);
    stack.setAdapter(new ItemAdapter(this, R.layout.item, items));
  }

  private static class ItemAdapter extends ArrayAdapter<String> {
    public ItemAdapter(Context context, int textViewResourceId,
                       String[] objects) {
      super(context, textViewResourceId, objects);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View result=super.getView(position, convertView, parent);

      result.setBackgroundColor(0xFF330000 + (position * 0x0A0A));

      return(result);
    }
  }
}
```

(from WidgetCatalog/StackView/app/src/main/java/com/commonsware/android/wc/stack/MainActivity.java)

Visual Representation

This is what a `StackView` looks like in Android 4.0.3, based upon the sample app shown above:

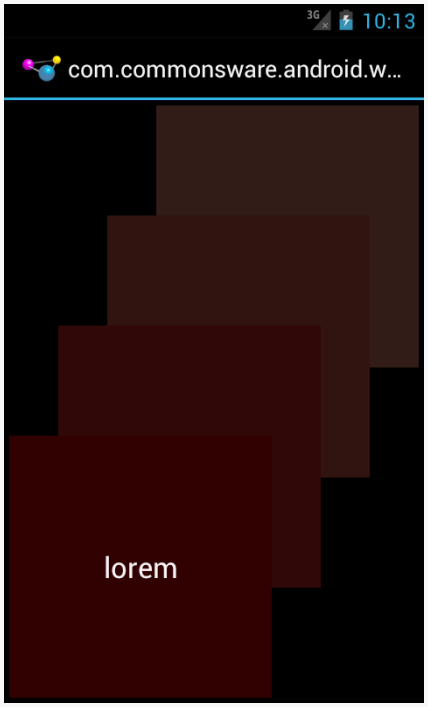


Figure 1091: Android 4.0.3, As Initially Seen

Widget Catalog: TabHost and TabWidget

Before we had the action bar and `ViewPager`, we had `TabHost` and `TabWidget` as our means of displaying tabs. Nowadays, in most cases, using tabs with a `ViewPager` is the preferred option. However, there may be cases where the classic tabs are a better solution, or you may have inherited legacy code that still uses `TabHost`.

Deprecation Notes

Just as `ListActivity` helps one use a `ListView`, `TabActivity` helps one use a `TabHost`. However, `TabActivity` is marked as deprecated. That is largely because its parent class, `ActivityGroup`, is deprecated. While you can still use `TabActivity`, it is no longer recommended. It also is not necessary, as there are ways to use `TabHost` and `TabWidget` without using `TabActivity`, as will be demonstrated later in this chapter.

Key Usage Tips

There are a few widgets and containers you need to use in order to set up a tabbed portion of a view:

- `TabHost` is the overarching container for the tab buttons and tab contents
- `TabWidget` implements the row of tab buttons, which contain text labels and optionally contain icons
- `FrameLayout` is the container for the tab contents; each tab content is a child of the `FrameLayout`

You load contents into that `FrameLayout` in one of two ways:

1. You can define the contents simply as child widgets (or containers) of the `FrameLayout` in a layout XML file you are using for the whole tab setup
2. You can define the contents at runtime

Curiously, you do not define what goes in the tabs themselves, or how they tie to the content, in the layout XML file. Instead, you must do that in Java, by creating a series of `TabSpec` objects (obtained via `newTabSpec()` on `TabHost`), configuring them, then adding them in sequence to the `TabHost` via `addTab()`.

The two key methods on `TabSpec` are:

- `setContent()`, where you indicate what goes in the tab content for this tab, typically the `android:id` of the view you want shown when this tab is selected
- `setIndicator()`, where you provide the caption for the tab button and, in some flavors of this method, supply a `Drawable` to represent the icon for the tab

Note that tab "indicators" can actually be views in their own right, if you need more control than a simple label and optional icon.

Also note that you must call `setup()` on the `TabHost` before configuring any of these `TabSpec` objects. The call to `setup()` is not needed if you are using the `TabActivity` base class for your activity.

A Sample Usage

The sample project can be found in WidgetCatalog/Tab.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/tabhost"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <TabWidget android:id="@android:id/tabs"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
      <AnalogClock android:id="@+id/tab1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
      />
      <Button android:id="@+id/tab2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="A semi-random button"
      />
    </FrameLayout>
  </LinearLayout>
</TabHost>
```

(from WidgetCatalog/Tab/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.tabhost;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TabHost;

public class TabDemo extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    TabHost tabs=(TabHost)findViewById(R.id.tabhost);

    tabs.setup();

    TabHost.TabSpec spec=tabs.newTabSpec("tag1");

    spec.setContent(R.id.tab1);
    spec.setIndicator("Clock");
    tabs.addTab(spec);

    spec=tabs.newTabSpec("tag2");
    spec.setContent(R.id.tab2);
```

```
    spec.setIndicator("Button");
    tabs.addTab(spec);
  }
}
```

(from WidgetCatalog/Tab/app/src/main/java/com/commonsware/android/tabhost/TabDemo.java)

Note that ordinarily you would use icons with your tabs, and so the second parameter to `setIndicator()` would be a reference to a drawable resource. This particular sample skips the icons.

Visual Representation

This is what a `TabHost` and `TabWidget` look like in a few different Android versions and configurations, based upon the sample app shown above.

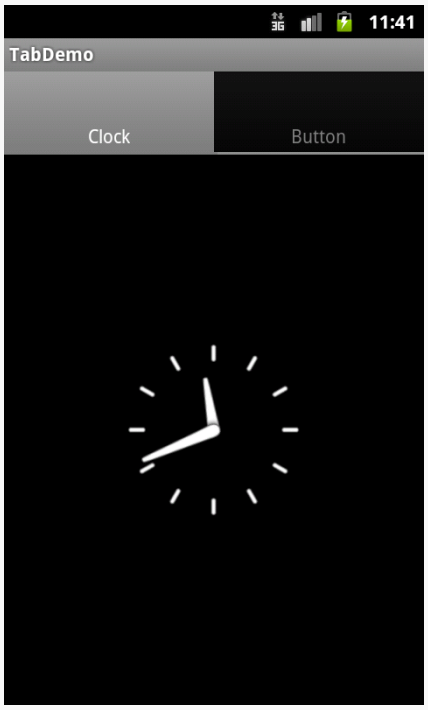


Figure 1092: Android 2.3.3

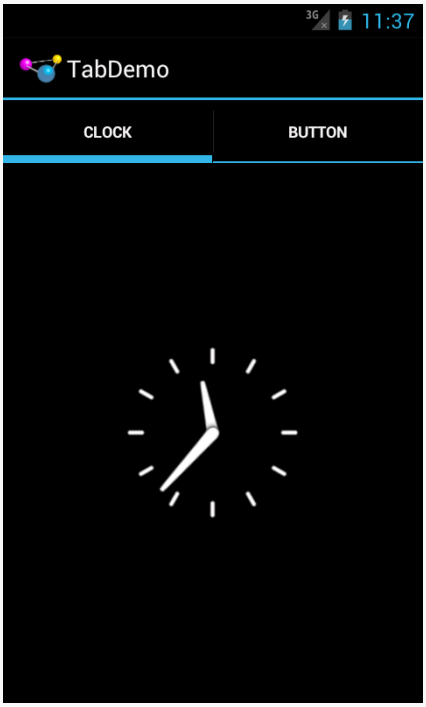


Figure 1093: Android 4.0.3

Widget Catalog: TimePicker

Just as DatePicker allows the user to pick a date, TimePicker allows the user to pick a time. This widget is a bit simpler to use, insofar as you do not have the option of the integrated CalendarView as you do with DatePicker. In other respects, TimePicker follows the patterns established by DatePicker.

Note that TimePicker only supports hours and minutes, not seconds or finer granularity.

Key Usage Tips

With DatePicker, the act of supplying an OnDateSetListener also required you to supply the year/month/day to use as a starting point. TimePicker is more intelligently designed: setting the OnTimeSetListener is independent from adjusting the hour or minute.

As with DatePicker, TimePicker works well with Calendar and GregorianCalendar, in terms of setting and getting the hour/minute/second from the TimePicker and converting it into something you can use in your code.

There is a bug in which your OnTimeSetListener is not invoked when the user changes between AM and PM when viewing the TimePicker in 12-hour display mode.

A Sample Usage

The sample project can be found in WidgetCatalog/TimePicker.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center_vertical">

  <TimePicker
    android:id="@+id/picker"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

(from WidgetCatalog/TimePicker/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.wc.timepick;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TimePicker;
import android.widget.TimePicker.OnTimeChangedListener;
import android.widget.Toast;
import java.util.Calendar;

public class TimePickerDemoActivity extends Activity implements
    OnTimeChangedListener {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TimePicker picker=(TimePicker)findViewById(R.id.picker);

    picker.setOnTimeChangedListener(this);
  }

  @Override
  public void onTimeChanged(TimePicker view, int hourOfDay, int minute) {
    Calendar then=Calendar.getInstance();

    then.set(Calendar.HOUR_OF_DAY, hourOfDay);
    then.set(Calendar.MINUTE, minute);
```

```
    then.set(Calendar.SECOND, 0);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_SHORT)
         .show();
  }
}
```

(from WidgetCatalog/TimePicker/app/src/main/java/com/commonsware/android/wc/timepick/TimePickerDemoActivity.java)

Visual Representation

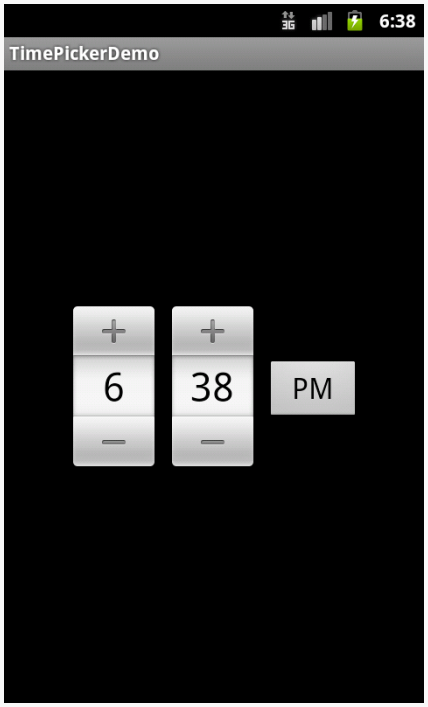


Figure 1094: Android 2.3.3

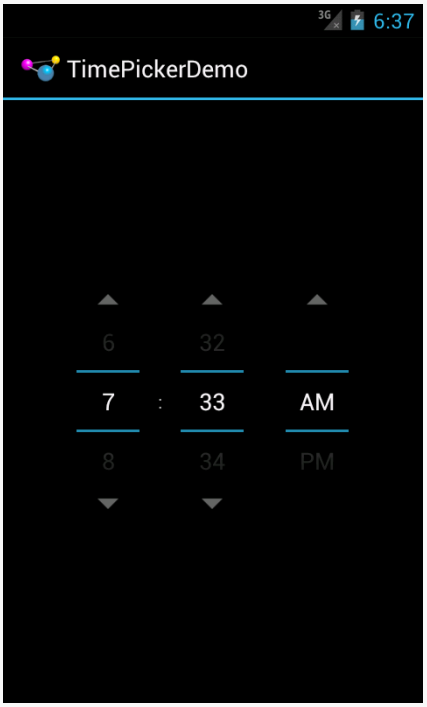


Figure 1095: Android 4.0.3

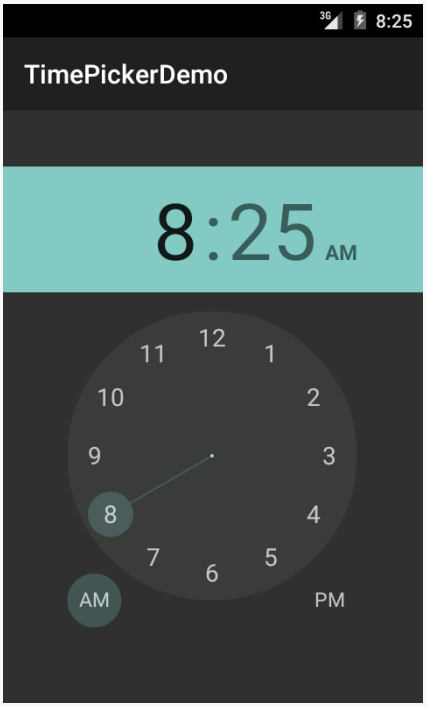
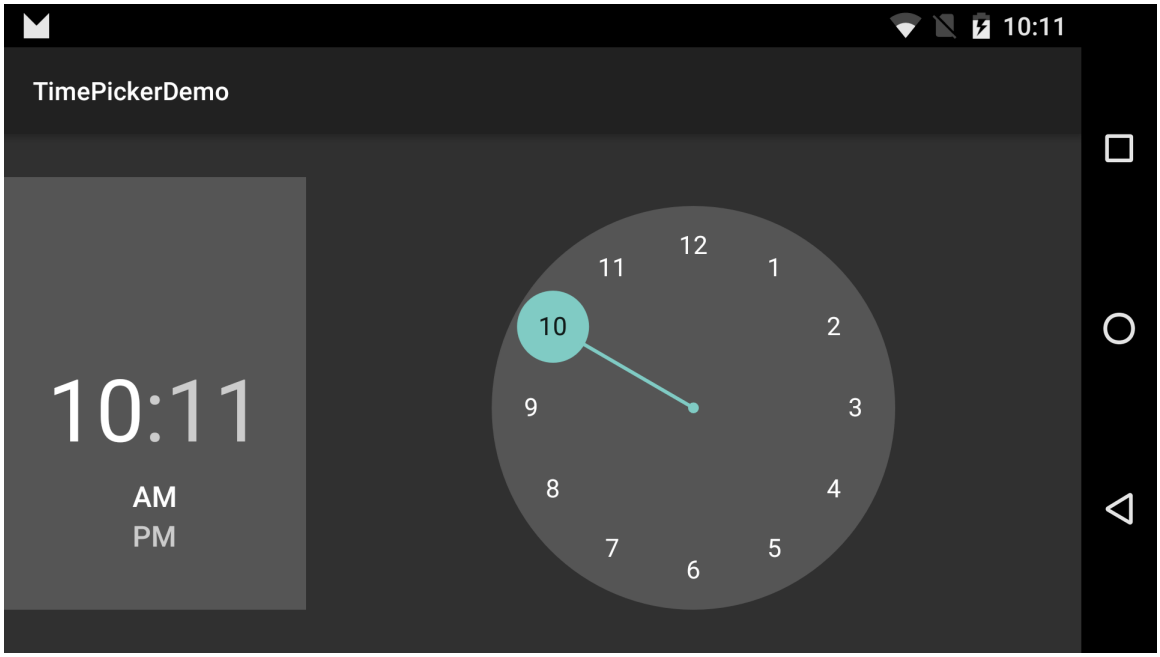


Figure 1096: Android 5.0


10:11
TimePickerDemo
10:11
AM
PM
12
1
2
3
4
5
6
7
8
9
10
11

Widget Catalog: ViewFlipper

A `ViewFlipper` behaves a bit like a `FrameLayout` that is set up such that only one child can be visible at a time. You can control which of those children is visible, either by index or via `showNext()`/`showPrevious()` methods to rotate between them.

You can also set up animated effects to control how a child leaves and the next one enters, such as applying a sliding effect.

And, you can set up `ViewFlipper` to automatically flip between children on a specified period, without further developer involvement. This, coupled with the animation, can be used for news tickers, ad banner rotations, or the like where light animations (e.g., fade out and fade in) can be used positively.

Key Usage Tips

`ViewFlipper` can have as many children as needed (within memory constraints), though you will want at least two for it to be meaningful.

By default, the transition between children is an immediate "smash cut" — the old one vanishes and the new one appears instantaneously. You can call `setInAnimation()` and/or `setOutAnimation()` to supply an `Animation` object or resource to use for the transitions instead.

By default, the `ViewFlipper` will show its first child and stay there. You can manually flip children via `showNext()`, `showPrevious()`, and `setDisplayedChild()`, the latter of which taking a position index of which child to display. You can also have automatic flipping, by one of two means:

1. In your layout, `android:flipInterval` will set up the amount of time to display each child before moving to the next, and `android:autoStart` will indicate if the automated flipping should begin immediately or not
2. In Java, `setFlipInterval()` serves the same role as `android:flipInterval`, and you can control when flipping is enabled via `startFlipping()` and `stopFlipping()`

A Sample Usage

The sample project can be found in WidgetCatalog/ViewFlipper.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <ViewFlipper android:id="@+id/details"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  </ViewFlipper>
</LinearLayout>
```

(from WidgetCatalog/ViewFlipper/app/src/main/res/layout/main.xml)

Activity:

```
package com.commonsware.android.flipper2;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.ViewFlipper;

public class FlipperDemo2 extends Activity {
  static String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
                         "consectetuer", "adipiscing", "elit",
                         "morbi", "vel", "ligula", "vitae",
                         "arcu", "aliquet", "mollis", "etiam",
                         "vel", "erat", "placerat", "ante",
                         "porttitor", "sodales", "pellentesque",
```

```
                          "augue", "purus"};
  ViewFlipper flipper;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    flipper=(ViewFlipper)findViewById(R.id.details);

    for (String item : items) {
      Button btn=new Button(this);

      btn.setText(item);

      flipper.addView(btn,
                      new ViewGroup.LayoutParams(
                              ViewGroup.LayoutParams.FILL_PARENT,
                              ViewGroup.LayoutParams.FILL_PARENT));
    }

    flipper.setFlipInterval(2000);
    flipper.startFlipping();
  }
}
```

(from WidgetCatalog/ViewFlipper/app/src/main/java/com/commonsware/android/flipper2/FlipperDemo2.java)

Visual Representation

There is no visual representation of a `ViewFlipper` itself, as it renders no pixels on its own. Rather, it simply shows the current child.

Trail: Device Catalog

Device Catalog: Chrome and Chrome OS

Ever since Android and Chrome were moved under the same executive within Google, rumors abounded that Android and Chrome OS would merge in one form or fashion.

In 2015, Google started down that path, offering the ability for developers to start packaging Android apps to run on Chrome OS. And — albeit via a different mechanism — some Chrome OS devices now offer the Play Store and users can install compatible apps from there.

The exact number of Chromebooks that have been sold is subject to some debate. One analyst pegged business (B2B) Chromebook sales in the first half of 2015 at around 2 million, with an upbeat, pro-Chromebook spin. Another analyst indicated that total sales for Chromebooks in 2014 were 6 million, a tiny percentage of PC/laptop sales. In mid-2016, IDC estimated that Chromebook sales for the first quarter of 2016 were around 2 million, exceeding the sales of Apple's line of Mac notebooks. But one analyst is predicting 17 million Chromebooks to be sold in 2023, which suggests that growth will be modest.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

How This Works

From the user's standpoint, Android apps appear alongside their Chrome OS counterparts. For example, the Play Store will be in their app launcher:

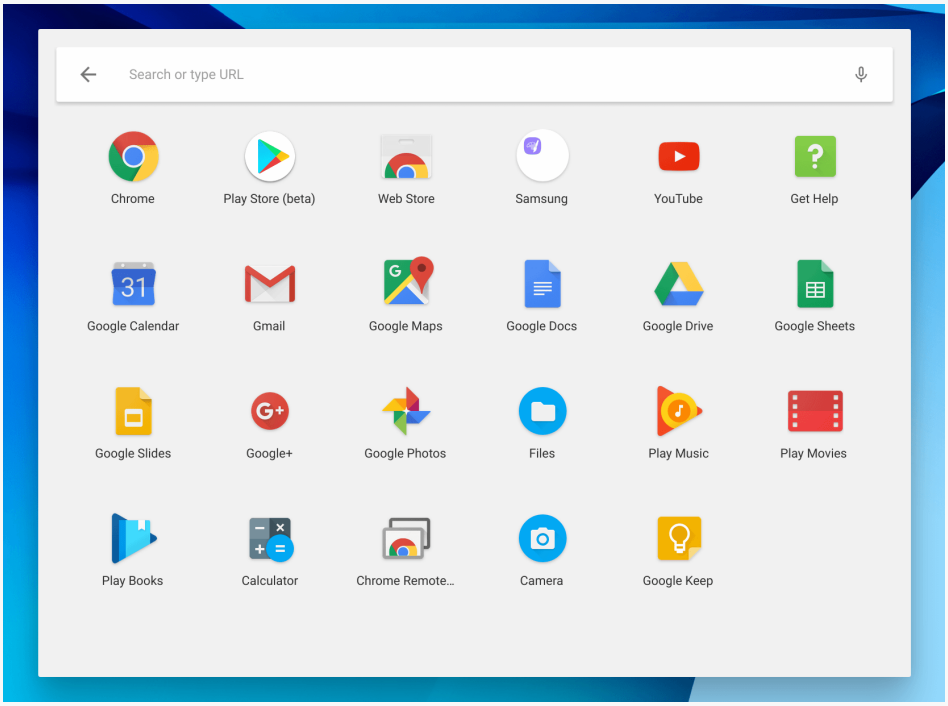


Figure 1097: Chrome OS App Launcher/Finder, As Initially Launched

Android apps appear in floating windows, similar to their Chrome OS counterparts:

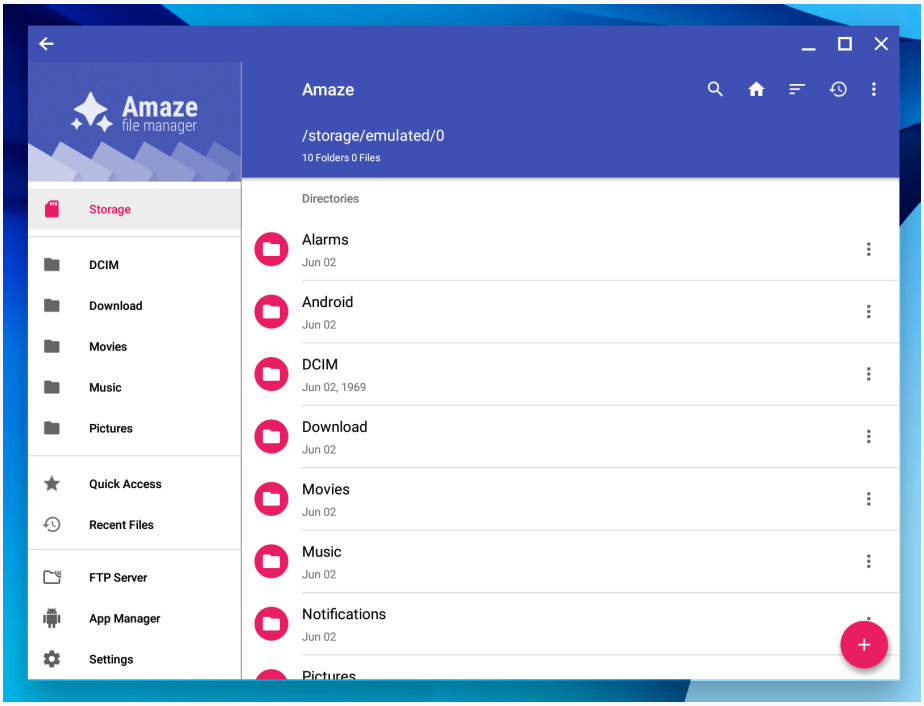


Figure 1098: Chrome OS, Showing Amaze File Manager

Right now, these windows cannot be arbitrarily resized. The user can switch between a floating window and a maximized view, via the title bar decorations in the upper-right corner. However, in the future, users should be able to resize these windows as they see fit, the same way they can do with native Chrome OS windows.

The Android apps see what resembles a normal Android environment. Right now, it is based on Android 6.0.1. Some aspects of the environment are shared between operating systems — most notably, the `Downloads` directory is accessible via external storage in Android and via the Files app in Chrome OS.

Testing Your App on Chrome OS

At the present time, there is limited documentation on how to test your Android app on Chrome OS. This is unfortunate.

The material in this section is derived from various Stack Overflow answers, Google+ posts, occasional bits of actual documentation, other random data sources, and a lot of experimentation.

Step #1: Get a Compatible Chrome OS Device

All Chrome OS devices shipped in 2017 and beyond are supposed to ship with Android support. Select older devices will get Android support as well. This page lists the official status of Chrome OS for various devices. Any device not listed there is unlikely to get Chrome OS, and even many of the "Planned" ones might drop off.

You will need to obtain one of these devices for testing. There is no Chrome OS emulator, let alone one that has Android app support.

Step #2: Enable Android Apps

If you want to test your app in the Play Store, it may be already enabled for you, in which case you can just install your app and try it out.

If it is not already enabled, but the device is already shipping in production form with Android support, you may need to enable it in Chrome OS's Settings app. To do that:

- In the lower-right corner of the Chrome OS desktop, click on the bar that shows the time, battery level, and your account picture:

Figure 1099: Chrome OS Status Bar

- You should then see a popup panel with status information and a few controls, akin to elements of the notification shade in Android:

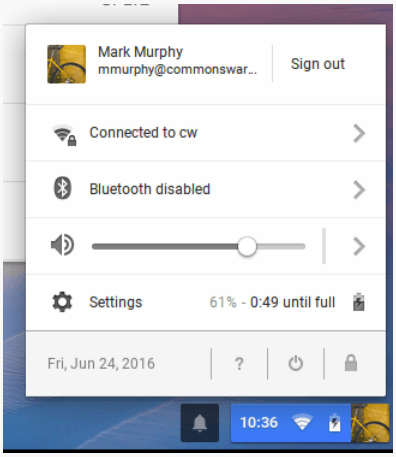


Figure 1100: Chrome OS Configuration Panel

The exact look of this panel will vary by device and Chrome OS version.

- In that panel, tap on "Settings"

The exact look of the Settings window will *also* vary by device and Chrome OS version.

Towards the bottom of this Settings page, you should see an "Android Apps" or "Google Play Store" section:

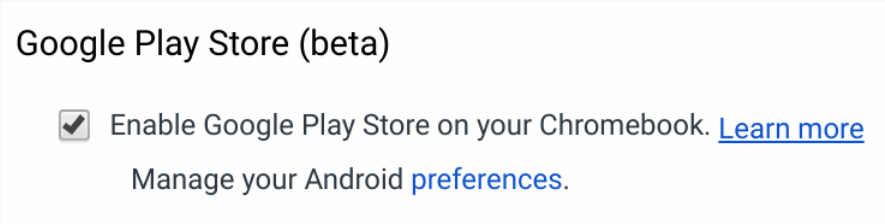


Figure 1101: Chrome OS Android Apps Checkbox

Click the "Enable Android Apps to run on your Chromebook" checkbox or the "Enable Google Play Store on your Chromebook" checkbox, whichever one you have. At this point, you should gain access to the Play Store and be able to install apps from there.

Also, note the "App Settings" or "Manage your Android preferences" link below that checkbox. This will bring up the *Android* Settings application, which is separate and distinct from the Chrome OS Settings page that you are on. There is no icon for the

Android Settings app in the regular Chrome OS app launcher; you have to know to come here to adjust the Android settings.

Step #3: Switch the Device to the Dev Channel

Some of the devices listed on that Chrome OS/Android status page indicate that the status is "Stable Channel". Such devices either ship with Android support or will get it with a regular Chrome OS update, and so the instructions in the preceding section should be sufficient for you to test a production app.

For devices whose status is "Beta Channel" or "Dev Channel", you will need to switch your Chrome OS device to that channel. This works much like the "canary channel" for Android Studio releases, or the dev channel for Chrome/Chromium releases. It configures the device to pull from a different update source, one that pushes updates more aggressively than it does to normal users.

To do this:

- In the "Settings" page, click on the "About Chrome OS" link
- In the About dialog, click on "Check for and apply updates", to make sure that you are on the latest stuff for your current channel:

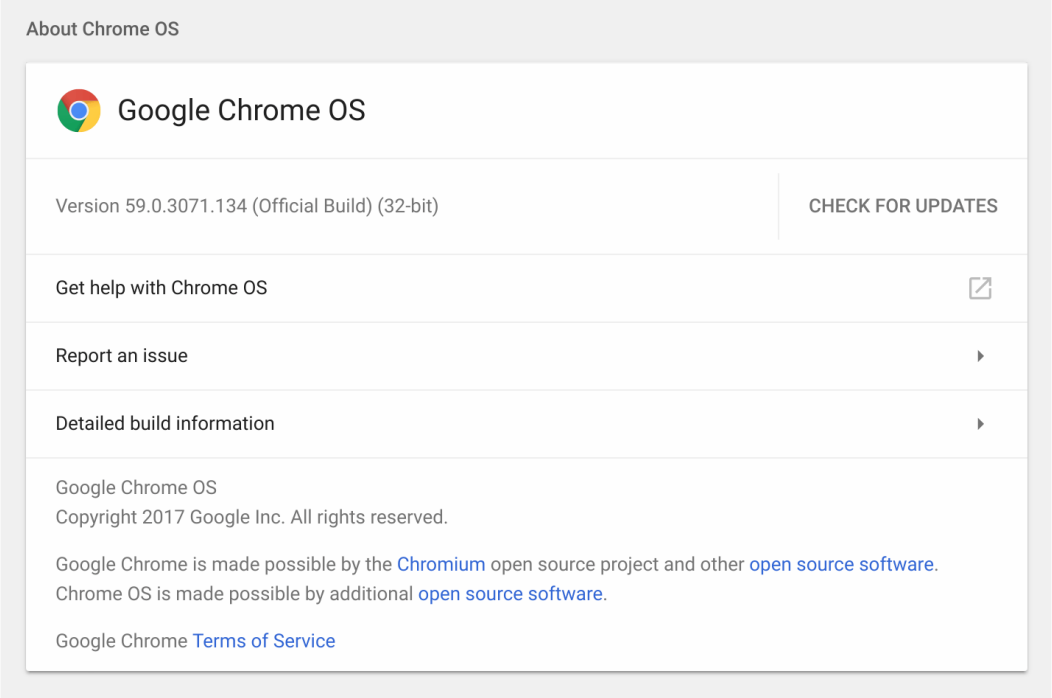


Figure 1102: Chrome OS About Screen

- Once that is done (and you have returned to the About dialog after a reboot, if updates were needed), click the "Detailed build information" link, which brings up a bunch of details regarding your Chrome OS device:

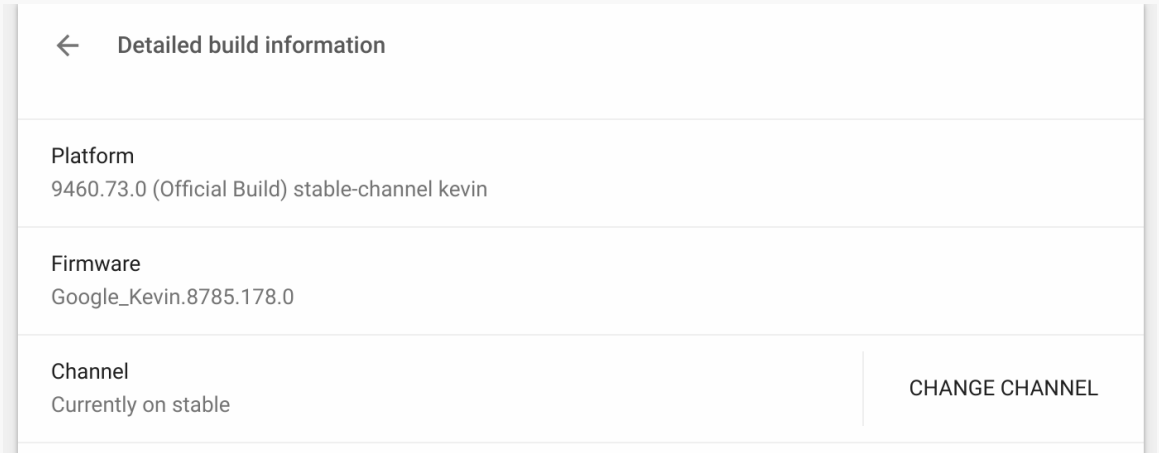


Figure 1103: Chrome OS Detailed Build Information Screen

- Click on the “Change channel...” button in the Channel category, which brings up a dialog showing available channels:

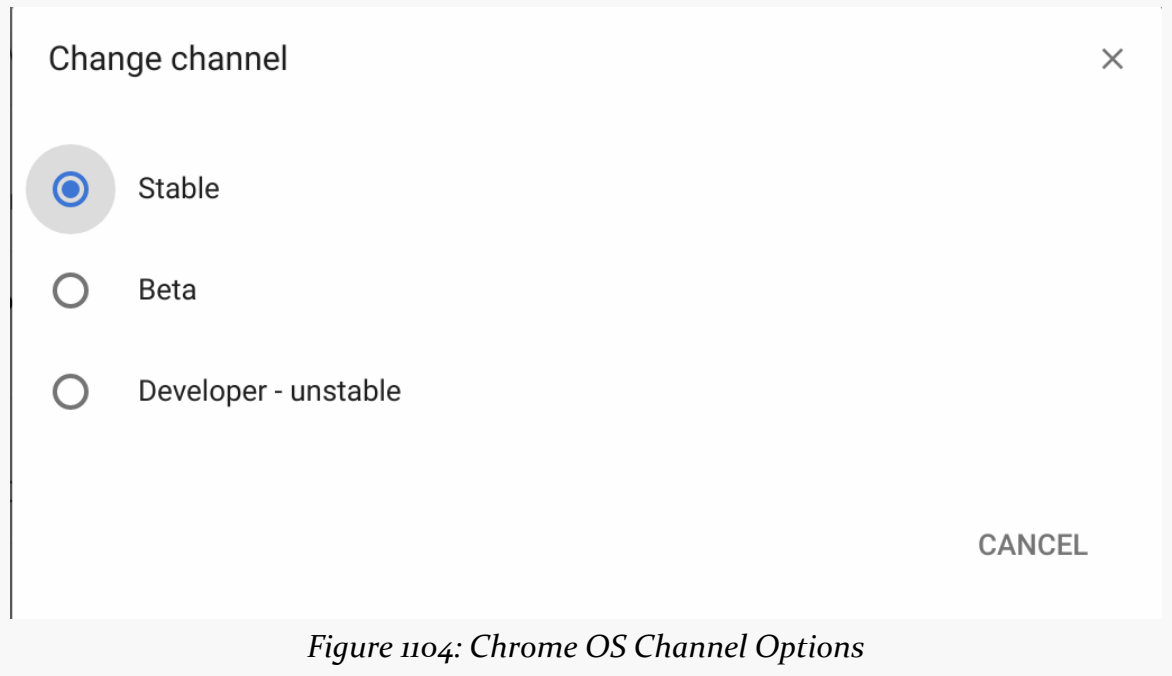


Figure 1104: Chrome OS Channel Options

- Click on the radio button for your desired channel and accept the dialog

This should apply a new round of updates, pulled from your chosen channel, then require you to reboot the Chrome OS device.

Step #4: Enable Chrome OS Developer Mode

Those steps are sufficient to allow you to download and run apps from the Play Store. For seeing if your already-shipping app works on Chrome OS, that may be sufficient. However, you have no access to Logcat, and you have no means of running debug builds or otherwise testing anything *other* than your already-shipping app.

For that sort of stuff, you need to enable Chrome OS developer mode. There is no direct analogue for this in the Android world. The closest match is enabling `fastboot`, perhaps as part of installing an Android developer preview ROM or some other custom ROM.

Unfortunately, the instructions for enabling developer mode are device-specific and arcane.

The Asus Chromebook Flip and the 2015 edition of the Chromebook Pixel share the same instructions:

- First, find the hardware power button. On the Flip, this is on the left edge towards the front.
- Next, shut down the device normally.
- Then, hold down the Esc and Refresh keys, and while holding them down, press the power button to turn on the device. The Refresh key looks like a circular arrow and may be denoted as F3 on your keyboard. This brings up a "recovery screen".
- On the recovery screen, press Ctrl-D. This screen should prompt for confirmation, then reboot the device into developer mode.

From this point forward, your device will be developer mode. However, on subsequent reboots, now that recovery screen will always appear. There, you have three choices:

1. Wait 30 seconds, in which case the device will continue its boot into developer mode
2. Press Ctrl-D to skip the 30-second wait
3. Follow the instructions on the screen to leave developer mode and return to normal operation

Step #5: Set Up the Android Environment

At this point, you will need to go through some standard steps for doing development in an Android environment, via the Android edition of the Settings app. You get to this via that "App Settings" or "Manage your Android preferences" link in the "Android Apps" (or "Google Play Store") category of the Chrome OS Settings page.

There, you can:

- Go into "About" and tap seven times on the "Build number" entry to enable developer settings
- Go into "Developer options" and enable USB debugging (and anything else that you typically use)
- Go into "Security" and enable "Unknown sources"

Note that those options may already be enabled, once you enable "Developer options" via the seven-taps technique.

Step #6: Side-Load and Install Your App

At this point, you can try out custom builds of your app, by installing them manually (a.k.a., "side-loading").

Unfortunately, Chrome OS knows nothing about APK files, so you cannot use the Chrome OS Files app to install an APK file.

The simplest way to side-load apps is to have an Android app do it:

- a cloud storage client (e.g., Dropbox)
- an Android-native Web browser
- a file manager, to load an APK you got onto the device by some other means (e.g., Amaze File Manager)

As mentioned earlier, the `Downloads` folder in Chrome OS is shared with the same folder on external storage in Android. So, you can also copy an APK over via a USB flash drive, put it in `Downloads` using Chrome OS, then use a file manager to install it from `Downloads`.

Step #7: Get adb Working

However, you still do not have access to Logcat, or any ability to use development tools like Android Studio to work on apps on the Chrome OS device. Setting up `adb` access is possible but a bit complicated, based on the official instructions.

Step #7a: Configure the Chrome OS Device

Press Ctrl-Alt-T to open `crosh`, a quasi-shell provided in Chrome OS. At that command prompt, run the `shell` command to get to `bash`, a full Linux-style shell.

Then, execute the following statements:

```
sudo crossystem dev_boot_signed_only=0
sudo /usr/libexec/debugd/helpers/dev_features_rootfs_verification
sudo reboot
```

This will reboot your Chrome OS device. If you run into errors running the `dev_features_rootfs_verification` command, try rebooting after the `crossystem`, then dropping back into the shell and running the `dev_features_rootfs_verification` command again.

Then, go back into `bash` via `crosh`, and execute:

```
sudo /usr/libexec/debugd/helpers/dev_features_ssh
```

This enables an SSH daemon and presumably makes corresponding adjustments to the `iptables`-based firewall.

NOTE: You will need to execute this `dev_features_ssh` script after every reboot of your Chrome OS device. Whereas the other portions of this step persist after a reboot, this portion does not. However, after a firmware update, you may need to do all of these steps.

Step #7b: Find Your Chrome OS IP Address

Go into Settings (e.g., tap on the time/WiFi/battery/account bar in the lower-right, then tap the gear icon). Towards the top of the Settings page, there will be an "Internet connection" section.

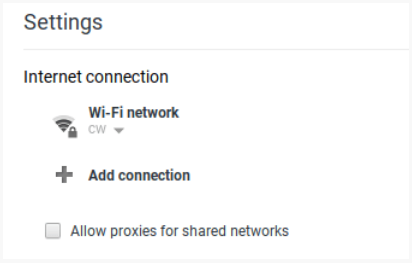


Figure 1105: Chrome OS Settings, Internet Connection Section

If you are using WiFi, tap on the "Wi-Fi network" item, then tap on the WiFi network that you are using. That should bring up a three-tab dialog with details about this network connection.

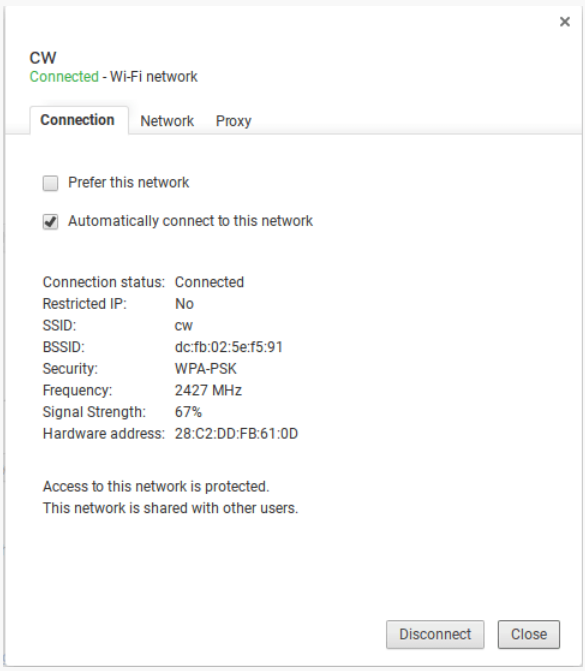


Figure 1106: Chrome OS Settings, Internet Connection Dialog

The middle tab — Network — will show your IP address:

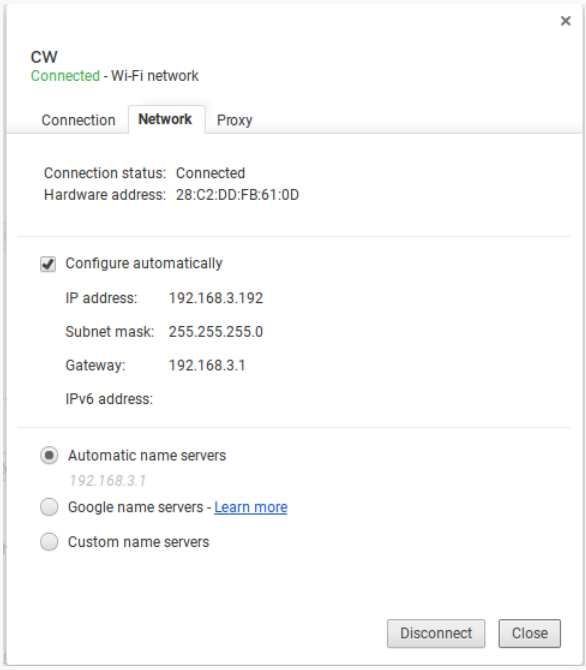


Figure 1107: Chrome OS Settings, Internet Connection Dialog, Network Tab

Make note of this address.

Step #7c: Connect to Chrome OS for Development

When you want to develop using the Chrome OS device as the target, execute `adb connect [IP]:22`, where `<IP>` is the same IP address that you used previously. Note that the official docs drop off the `connect` part, which does not work.

This should trigger the standard Android debugging authorization dialog, akin to what you see when you first try USB debugging on a phone or tablet. You will need to accept this dialog before continuing.

At this point, `adb devices` should show your connection, and you should be able to run apps on the Chrome OS device akin to how you do so for locally-connected devices, emulators, etc.

If the Chrome OS device goes to sleep, it will disable its WiFi connection. You will need to wake up the device and run `adb connect` again to be able to work with it from Android Studio.

If you reboot the Chrome OS device, you will need to agree once more to the Play Store terms of service and (try) to sign into the Play Store. The Play Store UI may appear to get stuck, with a never-ending progress bar. However, once you have gotten to that point, you should be able to use `adb connect` to re-connect to the Chrome OS device.

Be Prepared To Be Wiped Out

Some dev channel updates will wipe out your Android environment, deleting all apps and files. Hence, do not store things on the developer preview devices that you will regret losing. Also, you will need to re-do Step #4, and possibly Step #6, to re-establish a developer environment.

For example, in June 2017, Google started releasing an update to the Android environment that replaced Android 6.0 with Android 7.1.1. This required redoing some of the above steps, such as the seven-taps technique to enable “Developer options”. However, installed apps were unaffected.

Enabling Your App for Chrome OS

You may find that you go through Step #1 and Step #2 above — thereby enabling Android apps for the Chrome OS device — and find that your app is not available on the Play Store.

If you want your app to appear on the Play Store for Chrome OS devices, make sure that you are not requiring any of the following via a `<uses-feature>` element:

- `android.software.input_methods` (for implementing your own IME)
- `android.software.app_widget` (for apps whose primary purpose is to publish an app widget)
- `android.software.live_wallpaper` (for apps whose primary purpose is to publish some live wallpaper)
- `android.software.home_screen` (for home screen replacement apps)

Also, while the developer preview devices all have touchscreens, not all Chrome OS devices do. Ideally, you test your app using a keyboard and mouse or trackpad and confirm that it works well without a touchscreen. Then, add this to your manifest:

```
<uses-feature
  android:name="android.hardware.touchscreen"
  android:required="false" />
```

This advertises that you are willing to support non-touchscreen environments and therefore can be installed on non-touchscreen Chrome OS devices.

Your App on Chrome OS

On the surface, Android apps will perceive Chrome OS as just another Android device. Other than the manifest entries listed above, there is nothing that you absolutely need to do in your app to run on Chrome OS.

That being said, Chrome OS is going to be somewhat different than a normal Android device. Those differences may be something that you want to try to take into account.

Environment

Chrome OS is presently running API Level 25 (Android 7.1), though it contains some elements of Android 8.0's multi-window functionality.

We can tell what the API level is by logging `Build.VERSION.SDK_INT`, along with other values, as seen in the Introspection/EnvDump sample app. This has a single activity, designed to collect a bunch of device data and dump it to Logcat:

```
package com.commonsware.android.envdump;

import android.app.Activity;
import android.app.ActivityManager;
import android.content.pm.FeatureInfo;
import android.content.res.Configuration;
import android.os.Build;
import android.os.Bundle;
import android.util.DisplayMetrics;
import android.util.Log;
import android.widget.TextView;

public class MainActivity extends Activity {
  private static final String TAG="EnvDump";
  private final StringBuilder buf=new StringBuilder();

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    logBuildValues();
    logSystemFeatures();
    logActivityManagerStuff();
    logDisplayMetrics();
    logConfiguration();

    TextView tv=(TextView)findViewById(R.id.text);

    tv.setText(buf.toString());
  }

  private void logBuildValues() {
    log("Build.VERSION.SDK_INT="+Build.VERSION.SDK_INT);

    log("Build.BRAND="+Build.BRAND);
    log("Build.DEVICE="+Build.DEVICE);
```

```
    log("Build.DISPLAY="+Build.DISPLAY);
    log("Build.HARDWARE="+Build.HARDWARE);
    log("Build.ID="+Build.ID);
    log("Build.MANUFACTURER="+Build.MANUFACTURER);
    log("Build.MODEL="+Build.MODEL);
    log("Build.PRODUCT="+Build.PRODUCT);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
      StringBuilder buf=new StringBuilder();

      for (String abi : Build.SUPPORTED_ABIS) {
        if (buf.length() > 0) {
          buf.append(',');
        }

        buf.append(abi);
      }

      log("Build.SUPPORTED_APIS=" + buf);
    }
    else {
      log("Build.CPU_API="+Build.CPU_ABI);
      log("Build.CPU_API2="+Build.CPU_ABI2);
    }
  }

  private void logSystemFeatures() {
    for (FeatureInfo feature :
        getPackageManager().getSystemAvailableFeatures()) {
      log("System Feature: "+feature.name);
    }
  }

  private void logActivityManagerStuff() {
    ActivityManager mgr=(ActivityManager)getSystemService(ACTIVITY_SERVICE);

    log("heap limit="+mgr.getMemoryClass());
    log("large-heap limit="+mgr.getLargeMemoryClass());
  }

  private void logDisplayMetrics() {
    DisplayMetrics dm=new DisplayMetrics();

    getWindowManager().getDefaultDisplay().getMetrics(dm);

    log("DisplayMetrics.densityDpi="+dm.densityDpi);
    log("DisplayMetrics.xdpi="+dm.xdpi);
    log("DisplayMetrics.ydpi="+dm.ydpi);
```

```
    log("DisplayMetrics.scaledDensity="+dm.scaledDensity);
    log("DisplayMetrics.widthPixels="+dm.widthPixels);
    log("DisplayMetrics.heightPixels="+dm.heightPixels);
  }

  private void logConfiguration() {
    Configuration cfg=getResources().getConfiguration();

    log("Configuration.densityDpi="+cfg.densityDpi);
    log("Configuration.fontScale="+cfg.fontScale);
    log("Configuration.hardKeyboardHidden="+cfg.hardKeyboardHidden);
    log("Configuration.keyboard="+cfg.keyboard);
    log("Configuration.keyboardHidden="+cfg.keyboardHidden);
    log("Configuration.locale="+cfg.locale);
    log("Configuration.mcc="+cfg.mcc);
    log("Configuration.mnc="+cfg.mnc);
    log("Configuration.navigation="+cfg.navigation);
    log("Configuration.navigationHidden="+cfg.navigationHidden);
    log("Configuration.orientation="+cfg.orientation);
    log("Configuration.screenHeightDp="+cfg.screenHeightDp);
    log("Configuration.screenWidthDp="+cfg.screenWidthDp);
    log("Configuration.touchscreen="+cfg.touchscreen);
  }

  private void log(String msg) {
    Log.d(TAG, msg);
    buf.append(msg);
    buf.append('\n');
  }
}
```

(from Introspection/EnvDump/app/src/main/java/com/commonsware/android/envdump/MainActivity.java)

The following sections review the three major areas of the `EnvDump` output.

Build Values

Value	Acer Chromebook R11 C738T
Build.VERSION.SDK_INT	25
Build.BRAND	google
Build.DEVICE	cyan_cheets
Build.DISPLAY	R66-10403.0.0 release-keys
Build.HARDWARE	cheets
Build.ID	R66-10403.0.0
Build.MANUFACTURER	google

Value	Acer Chromebook R11 C738T
Build.MODEL	Acer Chromebook R11 (CB5-132T / C738T)
Build.PRODUCT	cyan
Build.SUPPORTED_APIS	x86,armeabi-v7a,armeabi

Of note, there is nothing obvious in here that indicates that this is a Chrome OS device.

Note that the Acer Chromebook R11 C738T has an x86 CPU, and `Build.SUPPORTED_APIS` indicates that it supports native x86 NDK binaries, along with ARM ones (presumably via `libhoudini`).

System Features

Value	Acer Chromebook R11 C738T
android.hardware.audio.low_latency	N
android.hardware.audio.output	Y
android.hardware.audio.pro	Y
android.hardware.bluetooth	Y
android.hardware.bluetooth_le	Y
android.hardware.camera.any	Y
android.hardware.camera.front	Y
android.hardware.faketouch	Y
android.hardware.location	Y
android.hardware.location.network	Y
android.hardware.microphone	Y
android.hardware.screen.landscape	Y
android.hardware.screen.portrait	Y
android.hardware.sensor.accelerometer	Y
android.hardware.touchscreen	Y
android.hardware.touchscreen.multitouch	Y
android.hardware.touchscreen.multitouch.distinct	Y
android.hardware.touchscreen.multitouch.jazzhand	Y
android.hardware.type.pc	Y
android.hardware.wifi	Y
android.software.backup	Y
android.software.freeform_window_management	Y
android.software.print	Y
android.software.voice_recognizers	Y

Value	Acer Chromebook R11 C738T
android.software.webview	Y
com.google.android.feature.GOOGLE_BUILD	Y
com.google.android.feature.GOOGLE_EXPERIENCE	Y
org.chromium.arc	Y
org.chromium.arc.device_management	Y

Many common system features are represented here.

Other Values

Value	Acer Chromebook R11 C738T
heap limit	192
large-heap limit	512
DisplayMetrics.densityDpi	160
DisplayMetrics.xdpi	160.631
DisplayMetrics.ydpi	161.216
DisplayMetrics.scaledDensity	1.0
DisplayMetrics.widthPixels	412
DisplayMetrics.heightPixels	732
Configuration.densityDpi	160
Configuration.fontScale	1.0
Configuration.hardKeyboardHidden	1
Configuration.keyboard	2
Configuration.mnc	0
Configuration.navigation	1
Configuration.navigationHidden	2
Configuration.orientation	2

Of note, the heap limits are generous (192MB normally, 512MB for large heap).

Screen Size and Orientation

The user can toggle between three different states for your activity's window:

- regular landscape
- regular portrait
- full-screen (akin to pressing the "maximize" button on a desktop OS)

Your activity will undergo configuration changes, as you might expect, when the user switches between these.

Lifecycle Events

Lifecycle events behave more or less as you might expect:

- While your app's window is the focused window, your app is in the normal running state (i.e., `onResume()` has been called).
- If another window takes over the focus, but your window is still visible, your activity in that window will be paused, then resumed if the window regains the focus.
- If the user minimizes your window, you will be called with `onPause()` and `onStop()`, as your activity is no longer visible.
- If the user switches between the floating-window and the full-screen sizes, your activity will undergo a configuration change and, by default, be destroyed and recreated. Note that there seems to be a bug, as your activity goes through a spurious extra pause/resume cycle before settling down.

Chrome OS remembers your last window size (floating or full-screen). On the next launch of your app, you will return to that size.

One oddity: if another window takes over the full screen, even though your window is no longer visible, your activity is not stopped. It is paused — that will happen once the window loses the focus. The only time you are stopped is if the window is minimized (or the activity is being destroyed).

Touchscreen and Keyboard Input

As is noted earlier in this chapter, while the developer preview Chromebooks have touchscreens, not all Chrome OS devices will. Chromebooks will have trackpads; Chromeboxes will rely on mice. And pretty much all Chrome OS devices will have full keyboards.

For touchscreen-equipped Chromebooks, the full suite of gestures should be available to developers. However, for devices lacking a touchscreen, using Android apps becomes... interesting.

For example, to scroll a `ListView` or `RecyclerView`, you cannot simply drag a scrollbar, because there *is* no scrollbar. Even if you putter around and arrange for scrollbars to become visible, they do not respond to touch input. Instead, users need

to know to press a meta key (e.g., Ctrl) while "swiping" the list with the mouse to be able to scroll. This is somewhat easier with a trackpad, as a two-finger swipe will scroll the list.

However, you really need to consider optimizing your app for keyboards and mice, rather than assuming touchscreens or trackpads with Chrome OS.

NDK

The good news is that NDK binaries work.

At the time of this writing, only the ASUS Chromebook Flip supports the developer preview Android app environment. The Flip has an ARM CPU, and so Android will look to use ARM binaries.

The original ARC environment for Android apps on Chrome OS *always* used ARM NDK binaries, even on x86 CPUs. It is unclear if the new Android app support uses the same approach, or whether it will use x86 binaries on x86-powered Chrome OS devices.

Storage

Internal and external storage both work fine on Chrome OS. However, outside of debugging tools, you do not have any ability to work with the files themselves as a developer, much like how you are limited in accessing files that are part of an emulator image.

The exception, as noted earlier in this chapter, is the `Downloads` directory on external storage. This is also available to Chrome OS users via the Files app.

Note that the Storage Access Framework works, for things like `ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT`. You do not have access to removable storage, though, even through the Storage Access Framework.

Notifications

Notifications work... at least to some extent.

Their look and feel gets normalized to Chrome OS styling, so they will not look like standard Android notifications:

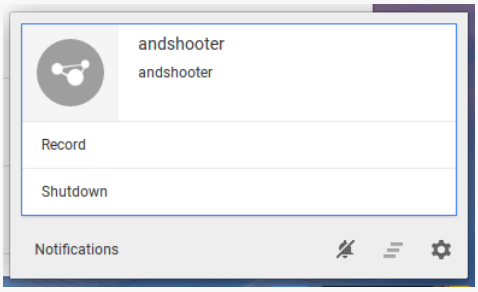


Figure 1108: Android Notification with Two Actions on Chrome OS

Also, actions seem to automatically cancel the `Notification`, even if that is not what you intended.

Internet Access

On the whole, HTTP access "just works", whether you are using `WebView` or `HttpUrlConnection`.

SSL also "just works", subject to the same sorts of limitations that you see in normal Android development, such as what root certificates are available for validating the SSL certificate for a particular `https://` URL.

The author has not tested lower-level socket operations at this time.

Miscellaneous Oddities

The user does not appear to be able to change the font scale. The Android Settings app lacks the "Display" screen where this would be controlled, and the Chrome OS font scale/zoom options have no effect on Android UIs.

`Theme.Translucent.NoTitleBar` does not work, insofar as you will still wind up with a window in Chrome OS, one with an all-black background:

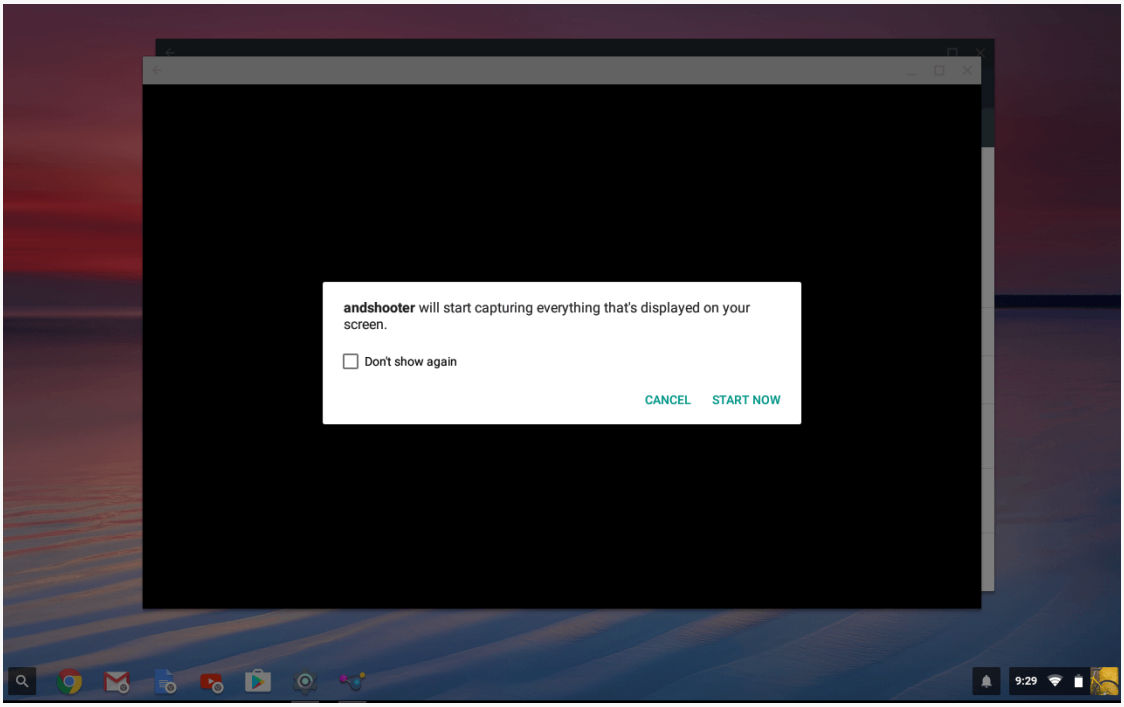


Figure 1109: "Invisible" Activity on Chrome OS

Android apps cannot work with external displays plugged into the Chrome OS device, using `Presentation` or similar techniques. It is possible that this is because some Chrome OS devices do not have their own built-in displays (e.g., ChromeBoxes) and rely on an external display as their primary display.

Distribution Options

Obviously, with the right manifest settings, you can distribute your app via the Play Store.

However, based upon the developer preview, ordinary users will not be able to flip the "Unknown sources" switch. That is limited to devices in developer mode. Hence, distribution by any other means (F-Droid, download from a Web browser, etc.) is not supported at this time.

Apps Sans Role

Many different sorts of apps will have no real role in a Chrome OS environment, such as:

- home screens and app-widget-only apps
- apps that primarily work in the background, as the user has to keep the app's window open for the process to remain running

Also, apps that go beyond the boundaries of the Android SDK, such as poking at various Linux elements inside of Android, will likely encounter problems as apps in Chrome OS. For example, three tested terminal emulators, for accessing the standard Linux shell in an Android environment, failed to run.

Getting Help

The official support point is the `#AndroidAppsOnChromeOS` hashtag on the Android developer Google+ community. There is an equivalent tag on Stack Overflow, which briefly was an official support point before being replaced by Google+.

If you encounter bugs, you are supposed to be able to file issues through this link on the Chromium issue tracker.

Android Things Basics

In December 2016, Google announced Android Things, a successor to the Brillo project, as Google's entry into the Internet of Things (IoT) market. Android Things allows you to write Android apps, akin to the ones that you write for phones and tablets, that can run on "headless" (no UI) devices like the Raspberry Pi 3.

This chapter will explore what Android Things is, how it differs from ordinary Android app development, and what you can do with it.

Conversely, this chapter will *not* cover details of connecting an Android Thing to any particular hardware (e.g., an external switch, an external LED, an external camera). The details for doing this will depend tremendously on the specific hardware, and so that aspect of Android Things is out of scope for this book. Instead, this book focuses on the classic Android side of Android Things, plus aspects that may be in common across Things scenarios (e.g., implementing secure remote control APIs, so the user can monitor and control a Thing via a mobile app or Web browser).

NOTE: This chapter is based off of Developer Preview 6.1 of Android Things. As a result, some of what is in here may change significantly with future previews and the first production-grade edition of Android Things. This chapter will be updated as needed over time to reflect those changes.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Um, What's a Thing?

The Internet of Things, generally speaking, represents embedded electronics that interact with the Internet that are not, in themselves, a general-purpose computing device.

That statement probably did not help much.

What's *Not* a Thing?

An Android phone is not considered to be an IoT device, as Android phones are designed to run arbitrary apps. The same goes for tablets and Android Wear devices. A desktop or notebook computer, equivalents of Android devices for other platforms (e.g., iOS), and kin are also general-purpose computing devices and not considered to be part of IoT.

Typical Things

Other items that connect to home/office networks, and perhaps the broader Internet, are more likely to be considered IoT devices. These include:

- so-called "IP cameras" (e.g., security cameras)
- thermostats (e.g., Nest)
- appliances (e.g., "smart refrigerators")
- home or office security systems

Home entertainment devices — "smart TVs", Internet-connected DVRs, and the like — sit somewhat in a middle ground. Roughly speaking, if the user can install apps on a device, that device would not be considered an IoT device. However, the lines here definitely get murky, and there are many things that you might want to do with Android Things that would be equally relevant for Android TV.

Android Things and IoT

IoT devices themselves run the gamut from tiny low-power items to, well, TVs and refrigerators. Even vehicles might be considered part of IoT.

Android itself started out running on fairly low-end hardware (e.g., 500MHz single-core CPUs, 192MB RAM, etc.). However, Android itself has grown over the years, and even those specifications exceed what some IoT platforms offer.

As such, expect to see Android Things on "higher-end" IoT devices, ones with sufficient hardware power to be able to run Android.

Headless, But No Horseman

With the advent of Android Wear, we started to envision Android devices with tiny touchscreens.

Android TV — and Google TV before that — gave us Android devices without touchscreens, relying on other forms of input. Chrome OS offers the same sort of potential, where devices may not have touchscreens.

With Android Things, not only might you not have a touchscreen, but you might not have a *screen*.

IoT devices use a wide range of user interfaces:

- Modern full-color capacitive touchscreens
- Monochrome resistive touchscreens (akin to the original Palm devices)
- LCD panels and mechanical buttons
- No UI at all (with the IoT device being controlled via some other piece of hardware), referred to as "headless"

This radically changes the sorts of apps that we will create. With Android Things, the "heavy lifting" will be done primarily by background components, primarily services. While there is still a role for activities, even on a headless Thing, most Things will not use one for much in the way of UI, unless your Thing happens to support a touchscreen.

Setting Up a Raspberry-Flavored Thing

The Android Things developer preview supports a few "boards", or hardware devices, at the time of this writing:

- Two versions of the NXP Pico
- NXP Argon
- Raspberry Pi 3

Of those, the easiest one for most developers to obtain will be the Raspberry Pi 3. For early experimentations with Android Things, the Pi should be a reasonable

choice. Products that *ship* with Android Things may well wind up using different hardware.

Getting a Pi

The Raspberry Pi 3 Model B is sold in many places around the world, such as Adafruit in the US. Used Pi 3 devices may be available through your typical used-hardware channels.

What you will want is:

- the Pi 3 board itself
- a case to put the Pi 3 in, so you do not accidentally fry your Pi by touching it while it is powered on
- an 8GB (or larger) micro SD card to use for storage (ideally Class 8 or higher), plus some way to work with it on your development machine (e.g., USB card reader)
- a micro USB cable and power supply, such as one you might have from an Android phone
- an HDMI-capable monitor and cable, at least for temporary use while setting up the Pi 3
- an Ethernet cable, at least for temporary use while setting up the Pi 3

You may already have some of these things or can borrow them from somebody.

Dealing with the Console

Originally, you could just download a ZIP archive containing the Android Things image for your platform, such as for the Pi 3. Alas, Google thrives on complexity, and so now, to get started with Android Things, you need to get started with the Android Things Console first.

What's the Android Things Console?

Google is focusing on business customers for Android Things, with an eye towards those businesses building hardware that uses Android Things as its operating system. Part of the "value add" is that Google offers what amounts to "OTA-as-a-service" (OTAAAS?), where updates to your software for the hardware, plus updates to the OS itself, can be pushed by Google to your hardware.

This is accomplished via the Android Things Console. To some extent, the Android Things Console is to Android Things-powered hardware as the Play Console is to Android apps.

You can visit the Android Things Console in your favorite Web browser. You will need to be logged into the appropriate Google account. Also, the first time that you visit the Console, you will need to agree to the Android Things SDK License Agreement and the Android Things Console Terms of Service. Read both documents carefully, as they put some fairly stringent limitations on what you can do with Android Things and its SDK. Also, you may not be allowed to enter into those agreements on behalf of your employer, where relevant.

Creating a Product

Once you agree to all of those legal terms, you will come to a page listing your products... except that you do not have any products just yet. Instead, you will see a "Create a Product" button:

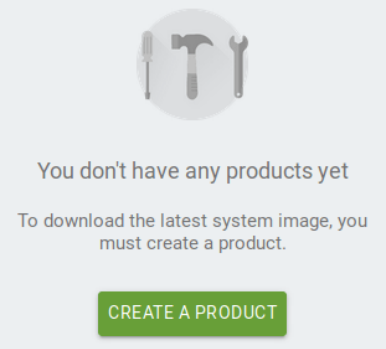


Figure 110: Android Things Console, As Initially Launched, With No Products

Clicking the "Create a Product" button brings up a dialog for you to provide details about the "product" that you are creating. Here, a "product" refers to a particular hardware-based product that you plan to use with Android Things.

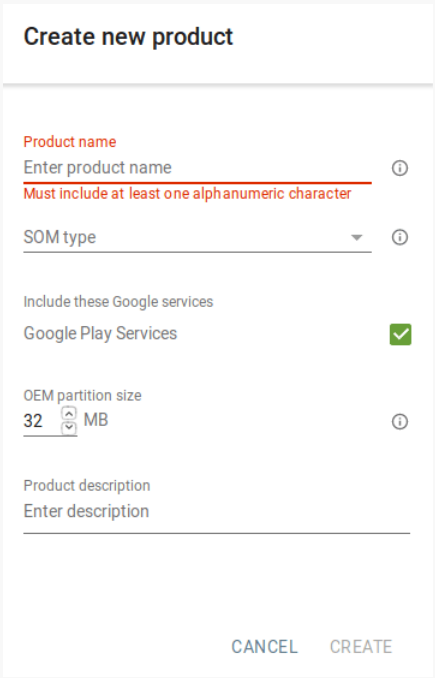


Figure 1111: Android Things Console, Showing New-Product Dialog

Here, you need to provide:

- Some name for the product, which will be used to identify it elsewhere in the Console
- The "SOM type", which is a fancy way of saying "hardware platform" (e.g., Raspberry Pi 3)
- Whether or not you want Google Play Services to be in the firmware or not
- How much size you want set aside for your app and its data ("OEM partition size"), which must be between 32MB and 512MB
- A short description of the product, which has no obvious use

Most likely, when you get started, you will want to create a "scrap" project that you can use just to get comfortable with the Console.

You can then click "Create" to create this product entry.

Configuring the Product

You will then be taken to a three-tab screen for your newly-defined product:

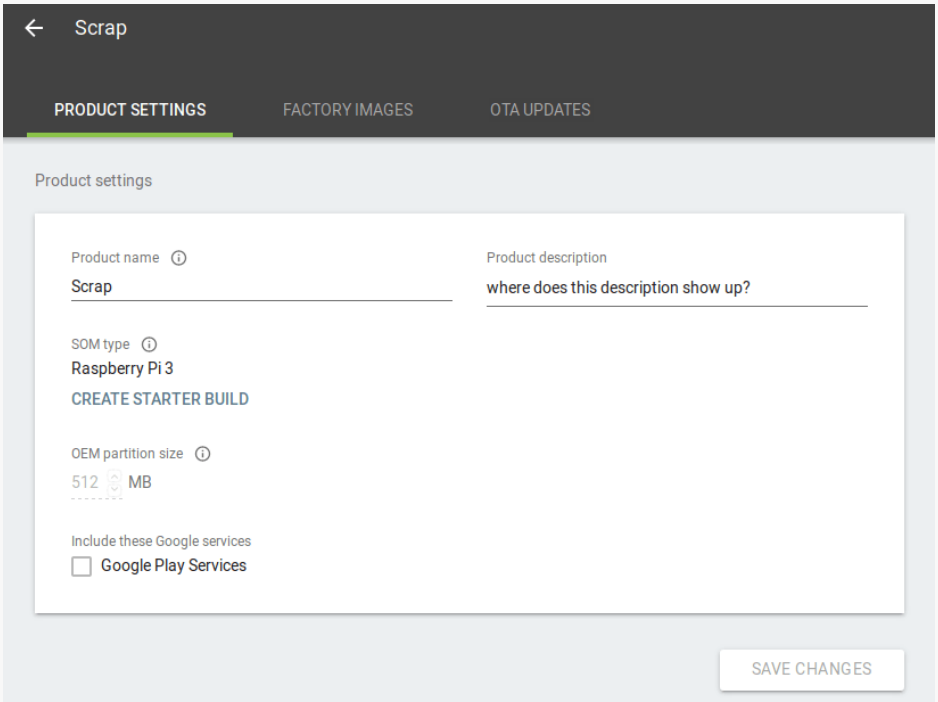


Figure 1112: Android Things Console, Showing Product Configuration Screen

Your next step is to click the "Create Starter Build" function, located in the middle of that form. This brings up an oddly-formatted wizard-ish UI for setting up a factory image for your product:

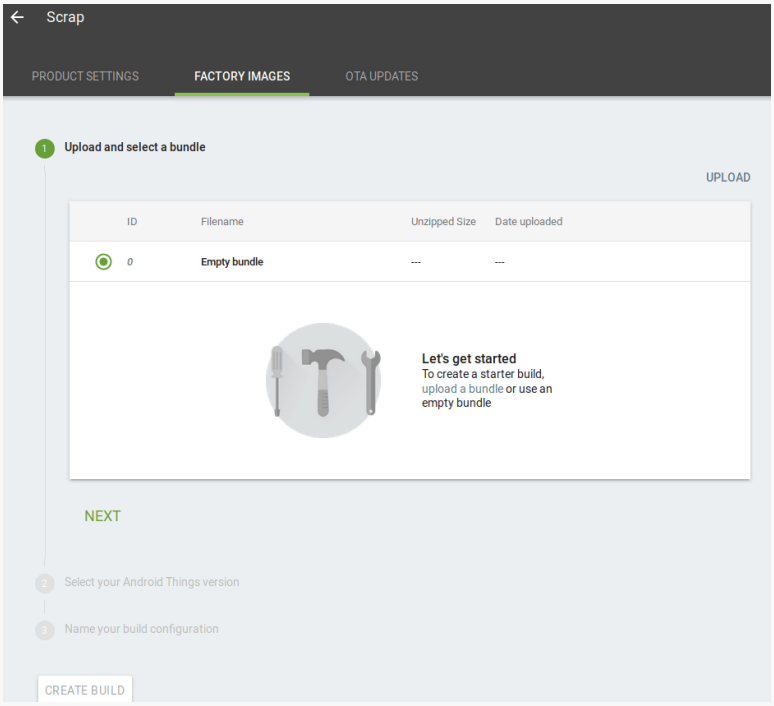


Figure 1113: Android Things Console, Showing Factory Image Wizard, First Page

What this screen refers to as a "bundle" is a mix of apps and other data that you want to pre-load onto the Android Things device that you will be building. At the outset — particularly for this scrap project — you will not have such a bundle yet. An "Empty bundle" bundle is pre-selected for you instead. So, just click "Next" to proceed to the next page of the wizard:

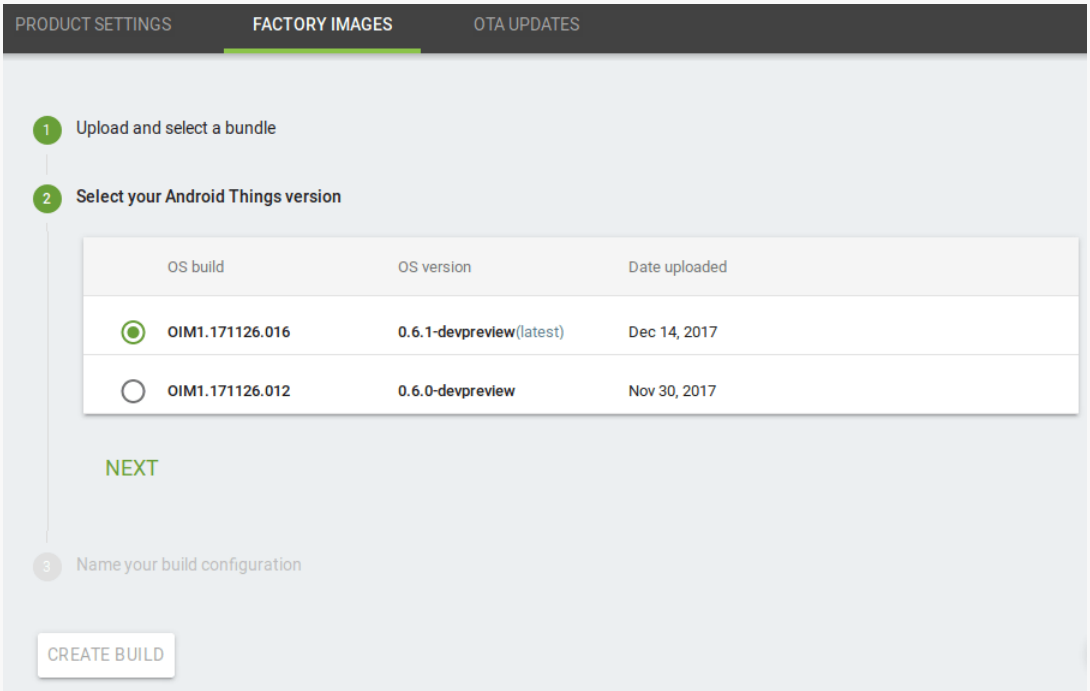


Figure 1114: Android Things Console, Showing Factory Image Wizard, Second Page

Here, you can choose from different versions of Android Things as your starting point, though you can only go back as far as Android Things 0.6. Once you have chosen your Things OS version, click Next to proceed to the third page of the wizard:

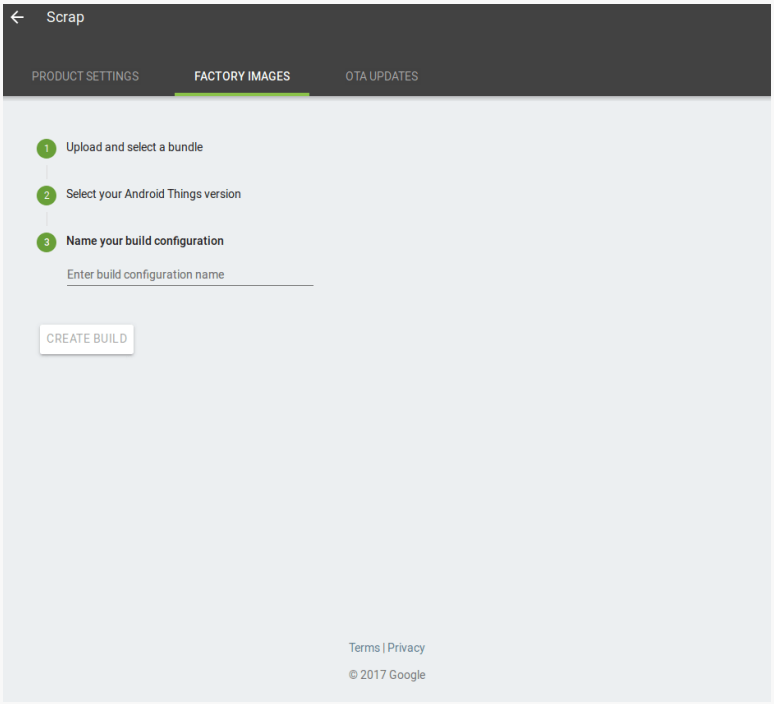


Figure 1115: Android Things Console, Showing Factory Image Wizard, Third Page

Here, you need to supply a "build configuration name". This will appear in the "Build configuration list", which is at the far bottom of the wizard:

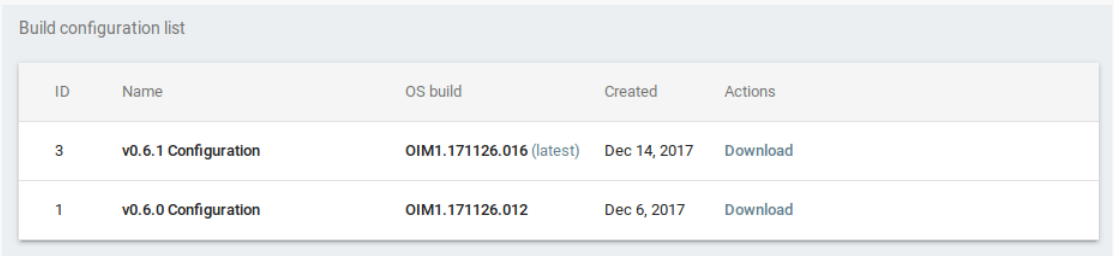


Figure 1116: Android Things Console, Showing Factory Image Wizard, Build Configuration List

Initially, the "Actions" column will show a progress indicator as your firmware image is being assembled. Eventually, it will turn into a "Download" link, which you can click to download the firmware image. That in turn will bring up a progress indicator while the download is "packaged". After a while, your browser's save-as dialog will appear to allow you to save the ZIP file containing the firmware.

Flashing the Image

What you get for your work in the Android Things Console is the same ZIP file that used to be just a simple download.

Next, you need to load that copy of Android Things onto the micro SD card that you want to use with your Pi 3. You have three options for doing this: Etcher, the Android Things Setup Utility, or manually.

Etcher

The Pi Foundation's recommended approach for installing a Pi images is through Etcher. This is an open source project for burning OS images to micro SD cards, bootable USB flash drives, and the like. Etcher is an Electron app, and so is available for Windows, macOS, and Linux alike.

Etcher makes installing your Android Things image fairly simple, via a three-step process:

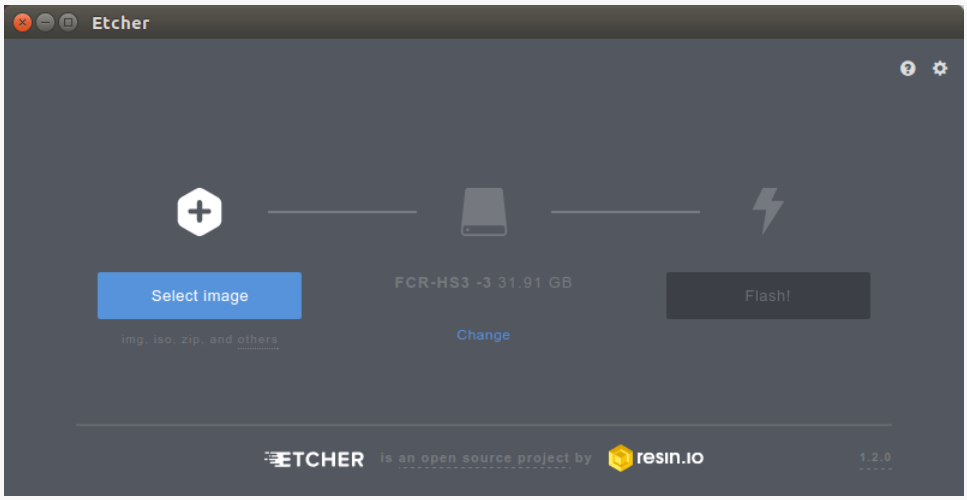


Figure 117: Etcher Main Screen

1. Selecting the image ZIP file
2. Selecting the flash drive onto which you want to install
3. Clicking "Flash" to flash the contents

This will take some time to complete, with a progress indicator and estimated time remaining clock to keep you informed of the progress.

Android Things Setup Utility

Google has their own similar tool to Etcher, called the Android Things Setup Utility. This is available from the "Tools" area in the Android Things Console, which you can get to via the "hamburger" nav drawer icon on any Console page.

However, the installation instructions require you to install this tool with administrator or superuser privileges, which is not a particularly sensible thing to do when alternatives exist.

Manually

The classic approach is for you to use your development machine's OS to unZIP that archive and copy it to the micro SD card:

- Linux
- macOS
- Windows

Booting and Connecting

At this point, you should be able to slide the micro SD card into the Pi and make basic connections:

- the monitor, via HDMI
- the Ethernet cable, connected to... something

One problem with Android Things is that you have no access to the Settings app, even if you plugged in a keyboard and mouse. Hence, you cannot set up WiFi without having `adb` access, and you cannot have `adb` access without a working network connection. So, at least for getting the Pi going with Android Things, you need to connect the Pi to a wired network, one that is reachable by your development machine. The second developer preview also provides support for using a USB to TTL serial cable as a way of accessing a command prompt, though the author has not tried this.

Finally, you can plug in the micro USB cable, connecting the Pi to a source of power. After a while, you should come to the default Android Things screen:

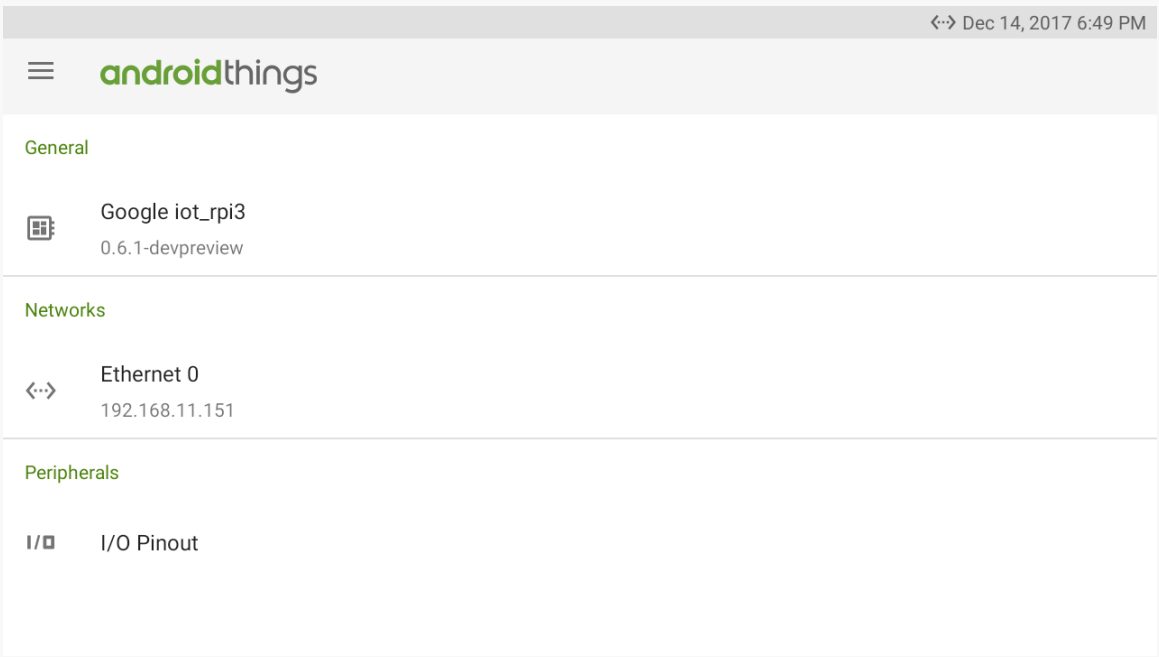


Figure 1118: Android Things Default Screen

For a while, you will see a green "Connect to Network" button. If you happen to have a mouse and keyboard connected to the Pi, you could use that to set up WiFi. If you have the Ethernet cable hooked up, that button eventually will be replaced with the IP address of your wired network connection.

If you were unable to obtain an HDMI monitor, but you do know the general IP address range of what the Pi should have been given, you can use tools like `nmap` to try to find your Thing:

```
nmap -p 5555,5554 192.168.11.2-254
```

This would search addresses from 192.168.11.2 through 192.168.11.254 for anything listening on the standard `adb` ports.

Enabling WiFi

If you want to use a WiFi connection instead of Ethernet, you can run the following `adb` command:

```
adb shell am startservice -n com.google.wifisetup/.WifiSetupService -a
WifiSetupService.Connect -e ssid ... -e passphrase ...
```

(NOTE: while that line will word-wrap due to its length, it should be entered all on one line)

The first `...`, for the `-e ssid` switch, is the SSID of your WiFi network. The second `...`, for the `-e passphrase` switch, is the passphrase to use to connect to your WiFi network, since everybody should be using a passphrase-protected WiFi access point. If, for inexplicable reasons, you do not have a passphrase on your access point, you can drop the `-e passphrase ...` part of the command, though on the whole you may be better off securing your access point.

At this point, in theory, your Android Things default screen should show a separate IP address for your WiFi connection. However, at least on Developer Preview 4, it will only show up on the default screen after a reboot. Logcat should show some `DhcpClient` debug messages that will have your IP address:

```
D/DhcpClient: Received packet: b8:27:eb:a1:e4:28 OFFER, ip /192.168.12.3, mask
/255.255.255.0, DNS servers: /192.168.12.1 , gateways [/192.168.12.1] lease time
7200, domain localdomain
```

Or, use `nmap` again, for your WiFi access point address range, to try to find your Thing.

If you have a hidden SSID, or if your passphrase contains certain special characters, there are additional command-line switches that you can use for those — see the documentation for details.

Powering Off Your Thing

While your production Thing might aim for continuous uptime, in development, you may need to power down the device (e.g., to flash a new developer preview image to the micro SD card). `adb shell reboot -p` works to power down the device.

`adb shell reboot`, without the `-p` switch, will reboot the Thing (e.g., to get the WiFi IP address to appear on the default screen).

An App For Your Thing

An Android Thing, by default, boots to the Things launcher screen as shown above. And, at that point, the Thing will sit there, sipping power, and otherwise not doing much.

This is not especially useful for the user of the Thing.

Presumably, your Thing has a purpose in life. And, that purpose is going to be embodied in at least one Android app that you load onto the Thing before shipping it to the user.

Setting up an app for Android Things is pretty much the same as setting up an app for ordinary Android devices. Here, we will focus on the bare minimum changes that you need to consider. Later, we will take a broader look at what's different for apps on Android Things.

The IOT_LAUNCHER

Android Things really does not want to show that launcher activity shown in the earlier screenshot. Android Things really wants to start some activity of yours. However, to do that, you need to advertise the activity that you want to be shown on boot, by having the right `<intent-filter>`:

```
<intent-filter>
  <action android:name="android.intent.action.MAIN"/>
  <category android:name="android.intent.category.IOT_LAUNCHER"/>
  <category android:name="android.intent.category.DEFAULT"/>
</intent-filter>
```

If Android Things finds an activity with that `<intent-filter>`, that activity, rather than the standard one, will be shown when the device boots.

The Regular Launcher

The problem with that `<intent-filter>` is that Android Studio and other development tools will not recognize it as anything special. So, for example, you might have to configure the run configuration for your module to direct Android Studio to your activity, so it knows which one to run.

However, there is nothing stopping you from having both the regular launcher `<intent-filter>` and the Things-targeted `IOT_LAUNCHER` `<intent-filter>`:

```
<activity android:name="MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
```

```
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.IOT_LAUNCHER"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

Now, you can run this normally from Android Studio without issue.

If you like, you could have the standard `<intent-filter>` in the `debug` source set, rather than in `main`, so your app does not ship with that `<intent-filter>`, but you can still use it in development mode. Or, you may actually want to ship with the standard `<intent-filter>`, particularly if your app will be targeting mobile devices, not just Android Things. Or, you might combine the `IOT_LAUNCHER` `<intent-filter>` with the `LEANBACK_LAUNCHER` `<intent-filter>` for an app that will ship to Android Things and Android TV.

The Library and the Library

There is a dedicated Things Support Library that provides APIs specific to Android Things, separate from other Android environments. In particular, it offers:

- a set of peripheral I/O APIs, so your app can communicate with hardware connected to your Thing, through a variety of standard I/O interfaces, including GPIO, I2C, PWM, SPI, and UART
- the User Driver API, which allows you to feed data into the Android framework that drives GPS, keyboard/mouse/controller events, and hardware sensors, for cases where Android Things does not have native integration with your desired hardware

Some apps will depend on these features. Others may not.

If your app needs this stuff, you will need to make some changes to your app to include this library.

First, you will need to add an entry in your `dependencies` in your module's `build.gradle` file, akin to how you add dependencies on Android Support Library modules or third-party libraries like OkHttp. However, this one will be a bit different, in that you will use the `compileOnly` keyword, instead of `implementation`:

```
dependencies {
    compileOnly 'com.google.android.things:androidthings:0.6-devpreview'
```

```
    // other dependencies here
}
```

`compileOnly` tells Gradle that the library should be used at compile time to satisfy the Java compiler, but that the contents of the library should *not* be packaged into the APK. Instead, the contents of that library will be "provided" to you automatically when your app runs on an Android Things device.

Also, you will need a `<uses-library>` element in your manifest, as an immediate child of the `<application>` element:

```
<uses-library android:name="com.google.android.things"/>
```

This tells Android that your app requires this firmware-supplied library, and the library should be loaded into your process when your process starts up.

If you plan on shipping your app to both Android Things and non-Things hardware, you can make `com.google.android.things` be optional:

```
<uses-library android:name="com.google.android.things" android:required="false"/>
```

Now, if the library is available, you will have access to it; if the library is not available (e.g., on ordinary Android devices), you will not have access to it. Without `android:required="false"`, Android would refuse to run your app on hardware lacking that library.

To find out if the library is available, for cases where you do use `android:required="false"`, use `Class.forName()` to see if `com.google.android.things.AndroidThings` exists. If `forName()` succeeds, you can use classes of yours that refer to classes out of this library. If `forName()` throws an exception, you need to avoid classes of yours that attempt to use this library.

Other Gradle Settings

The official instructions ask that you use a `compileSdkVersion` of 26 or higher. Most likely, that is not absolutely required. In general, though, you should be using that versions (or higher) for new app development anyway.

The UI

Of course, since we are now arranging for our activity to be launched when the device boots, it would help if that activity were actually useful.

If your device will have a display attached (touchscreen or purely output), then the UI that you show will be whatever you want to show the user. If your device will not have a standard display, then the activity UI is only for your purposes during development.

However, when your activity is visible, the standard Android Things activity is *not* visible... and that is where your IP addresses are listed. You will want to use `getInetAddresses()` on `java.net.NetworkInterface` to find out what IP addresses are known to the Thing, then include that information somewhere in your UI, so that you (and possibly the user) knows how to find this Thing. At least with Developer Preview 2, pressing Esc (i.e., BACK) from your `IOT_LAUNCHER` activity returns brings up the standard Android Things activity; pressing Esc again returns you to your `IOT_LAUNCHER` activity. It is unclear if this behavior will exist in a production environment, though.

Control Interfaces

An Android Thing is supposed to do something.

Presumably, the user needs to know if the Thing is doing what it is supposed to be doing. Sometimes, that can be determined, at least in part, by examining the environment. If a Thing is supposed to control a light, and the light is on and off when expected, the Thing is working. If a Thing is supposed to control a heater or air conditioner, and the temperature seems to be about right, probably the Thing is working. However, it might be nice to give the user more positive feedback that the Thing is doing what is expected. And, sometimes, it may be important to tell the user that the Thing is *not* doing what is expected, due to problems with the Thing or the hardware that it is supposed to be controlling.

Similarly, in many cases, the user needs to be able to configure the Thing. For a light-controlling Thing, the user needs to be able to control when the light is on or off, whether manually or via some programmed schedule. If the Thing controls room temperature, the user needs to be able to set the desired temperature range — and, again, that could be by configuring a schedule or performing some manual override.

And, in some cases, the user may need to help the Thing past some sort of problem (e.g., a battery-powered Thing needing to recharge its battery).

These scenarios imply some sort of control interface, where the user can see the Thing's current state and can manipulate that state, by one means or another.

Local

If your Thing is controlled via:

- A touchscreen connected to the Thing, or
- Some other type of screen (e.g., 2 line by 20 character LCD panel) or visual output (e.g., colored or labeled LEDs) and physical inputs (e.g., buttons, switches, knobs, sliders), or
- A hardware remote (e.g., TV remote)

then you have a local interface.

Local control interfaces, like the hardware that the Thing is supposed to control, depend a lot on the specifics of the hardware. Controlling LEDs is different than is controlling an LCD panel, which is different than controlling a touchscreen. As such, covering local control interfaces is out of scope for this book.

Remote

Conversely, a remote control interface is when the user is interacting with something else that affects the Thing's behavior, where the "something else" has its own smarts (more than does a TV-style hardware remote). This includes:

- using a Web browser
- using a mobile app (Android, iOS, etc.)
- using an end-user program for a desktop OS
- using a management program that controls lots of Things (e.g., all the lights for a warehouse, where each individual light is controlled by its own Thing)
- and so on

These can be further subdivided into how the browser, app, or program talks to the Thing: direct and indirect.

Direct

With a direct remote interface, the client (browser, app) talks to the Thing directly, with no intermediaries, other than perhaps local networking infrastructure (e.g., a WiFi access point).

Examples of this include:

- the Thing having a Web server operating on a specific port
- the Thing using other (non-Web) TCP/IP-based protocols as a way to send it commands, such as ZeroMQ
- the Thing allowing clients to pair with it via Bluetooth
- the Thing communicating with clients over WiFi Direct

Once set up, this can work well. However, security can become an issue, particularly in cases where the Thing is using TCP/IP (Web or something else) and the Thing winds up with a public IP address.

Also, these sorts of direct approaches tend to require relatively close access to the Thing, particularly if the Thing is *not* on the public Internet. So, for example, you could control your Thing-powered thermostat from within your home from your phone, but not be able to do so when you are not at home, as you cannot reach your Thing's Web server from other networks.

Indirect

Indirect remote control uses some sort of intermediary, so the Thing does not have to have an open TCP/IP port that might be exploited if the Thing winds up on the public Internet. These approaches also allow for wider access, while still keeping the Thing (relatively) secure.

Examples include:

- Using Firebase Cloud Messaging (FCM) to send messages from a client to the Thing and send back responses
- Using other "push messaging" systems, such as MQTT
- Using WebRTC, which typically leverages things like STUN servers to help connect two computers that cannot directly access one another
- Using STUN servers directly, bypassing WebRTC
- Using some home-grown proxy (e.g., durable WebSocket connection from the Thing to some hosted Web server that clients connect to)

These, however, intrinsically depend upon the Internet in most cases. That causes problems if the Internet is unavailable, whether permanently (e.g., a Thing deployed in a remote location) or temporarily (e.g., Internet outages, DDOS attacks affecting some of your infrastructure).

Also, control of the intermediary becomes an issue. If you use an intermediary managed by somebody else (e.g., Google in the case of FCM), you and users are dependent upon them continuing to offer the service. In the case where you control the intermediary, users have to trust that *you* will continue offering that service, as their Thing may become an Android-powered brick if you discontinue that service.

As a result, most likely, you will wind up supporting both a direct and an indirect mode of remote control, to help cover all the users' likely use cases.

What's Different?

So, with all that as prologue... what exactly is different about an Android Things environment, with respect to our apps and our jobs as developers, compared to with the phones and tablets that you are used to?

Preview Limitations

Some of the differences are tied specifically to the preview and may be addressed in future releases.

Many limitations are covered in the release notes, such as having no access to Bluetooth or USB APIs.

Of note to developers:

- If your app relies on `dangerous` permissions (e.g., `WRITE_EXTERNAL_STORAGE`), after your app is installed, you need to reboot your Thing before your app will be granted the permission in full. This is in lieu of the standard runtime permission logic, because requesting runtime permissions implies a user interface, and your Thing may not have a screen and such. The reboot is necessary both when your app using these permissions is installed the first time, plus any time you add a new `<uses-permission>` element for one of these permissions. This required reboot should go away in the future.

- If you have multiple apps, each with an `IOT_LAUNCHER` activity as described above, when your Thing boots, one of them gets started. Which one seems indeterminate.

UI

Android Things is targeting devices with no direct user interface, even though you create an `Activity` and can have a user interface if you want. As a result, certain capabilities that you might expect will not work.

For example, Android Things has no status bar. As a result, you have no means of showing a `Notification`. If you are using third-party libraries that might raise notifications, you could try implementing a `NotificationListenerService` to find out about those. That, in turn, will require some work to determine how to pre-register this service, as normally the user has to go into Settings to enable a `NotificationListenerService`.

Also, Android Things has no navigation bar, where HOME, BACK, RECENTS, etc. appear. Not having BACK can be a significant issue, if you are trying to present a local user interface. You will need to make sure that something in your UI (e.g., "up" affordance in the action bar) offers the user the ability to do what BACK ordinarily would do. Note that if you intend to have users attach some sort of keyboard (USB or Bluetooth) to your Thing, the Esc key maps to BACK, and the arrow keys and Enter can serve in the role of a D-pad, similar to how they behave in the Android SDK emulator.

At least at the present time, there is no input method editor enabled, which means you have no on-screen keyboard option for Things with touchscreens. If you plan on having the user connect a physical keyboard, this is not a problem. If you need the user to type something in on a touchscreen-enabled Thing, you will need to take some steps to enable an IME. Most likely, the details around this will change as Android Things evolves, and with a bit of luck, this step will not be required in the future.

`WebView` does not work at the present time, though this appears to be a temporary limitation. Dialogs and toasts, launched from an activity, work fine.

You are welcome to start other activities. Even though your `IOT_LAUNCHER` activity is "magic" with respect to what happens when the device boots, otherwise activities behave normally.

System

At present, Android Things runs Android 8.0 (API Level 26). Presumably, future updates will take on newer versions of Android itself.

However, not everything in Android 8.0 ships with Android Things. For example, the following system providers do not exist:

- `CalendarContract`
- `ContactsContract`
- `DocumentsContract`
- `DownloadManager`
- `MediaStore`
- `Settings`
- `Telephony`
- `UserDictionary`
- `VoicemailContract`

Activities that tie into those providers may not exist. There is no Downloads app, no dialer, no Contacts app, and so forth. Even the Settings app was removed in Developer Preview 6.

However, you *can* start activities from other apps, assuming that those activities exist. While you need to ship at least one APK with your Thing, containing your `IOT_LAUNCHER` activity, you can ship more than one APK if you so choose. Additional APKs work more or less as do their standard Android counterparts, including having activities. For example, the Launchalot sample app profiled elsewhere in the book works just fine, demonstrating that if desired, you could create a home screen with launcher activity icons and such.

Google Play Services exists, in part. As the documentation states, "As a general rule, APIs that require user input or authentication credentials aren't available to apps". So, while you could cast content to a Chromecast and use FCM for receiving push messages, you cannot buy things using Android Pay show ads via AdMob. It is likely that the mix of Play Services APIs that are available to Android Things will shift over time.

With respect to storage, internal and external storage appear to behave as normal. Since there is no UI, users have no means of granting permission to removable storage, and Android does not have an API for working with removable storage that does not require a UI. Hence, it is unclear what the plan is with respect to removable

storage. However, some developers have had success manually mounting USB storage.

`ACTION_BOOT_COMPLETED` seems to work as normal, giving your `BroadcastReceiver` control at boot time. This may not be necessary, given the `IOT_LAUNCHER` activity. However, if you have existing code that expects to use `ACTION_BOOT_COMPLETED`, it should work.

Since the device is continuously powered, there does not seem to be any effects from Doze mode. `AlarmManager` alarms fire regularly, without incident.

Environment Details

The `Introspection/EnvDump` sample project, as the name suggests, dumps a bunch of information about an Android environment to an on-screen `TextView`, plus Logcat.

Here is the output from a Raspberry Pi 3 running the Developer Preview 6:

```
Build.VERSION.SDK_INT=27
Build.BRAND=Things
Build.DEVICE=rpi3
Build.DISPLAY=iot_rpi3-userdebug 8.1.0 OIM1.171126.016 4498558 test-keys
Build.HARDWARE=rpi3
Build.ID=OIM1.171126.016
Build.MANUFACTURER=Google
Build.MODEL=iot_rpi3
Build.PRODUCT=iot_rpi3
Build.SUPPORTED_APIS=armeabi-v7a,armeabi
System Feature: android.hardware.ethernet
System Feature: android.hardware.bluetooth
System Feature: android.hardware.type.embedded
System Feature: android.hardware.usb.host
System Feature: android.hardware.screen.portrait
System Feature: android.hardware.bluetooth_le
System Feature: android.hardware.screen.landscape
System Feature: android.hardware.ram.normal
System Feature: android.hardware.lowpan
System Feature: android.software.webview
System Feature: android.hardware.camera.any
System Feature: android.hardware.location.network
System Feature: android.hardware.wifi
System Feature: android.hardware.location
System Feature: android.hardware.camera.external
```

```
System Feature: null
heap limit=256
large-heap limit=256
DisplayMetrics.densityDpi=240
DisplayMetrics.xdpi=0.0
DisplayMetrics.ydpi=0.0
DisplayMetrics.scaledDensity=1.5
DisplayMetrics.widthPixels=1280
DisplayMetrics.heightPixels=720
Configuration.densityDpi=240
Configuration.fontScale=1.0
Configuration.hardKeyboardHidden=2
Configuration.keyboard=1
Configuration.keyboardHidden=1
Configuration.locale=en_US
Configuration.mcc=0
Configuration.mnc=0
Configuration.navigation=1
Configuration.navigationHidden=2
Configuration.orientation=2
Configuration.screenHeightDp=456
Configuration.screenWidthDp=853
Configuration.touchscreen=1
```

The `android.hardware.type.embedded` system feature may be useful both for determining at runtime if you are running on Android Things (via `PackageManager` and `hasSystemFeature()`) or for limiting your app to only run on Android Things devices (via a `<uses-feature>` element in the manifest). For example, you could have:

```
private boolean isThing() {
  return(Build.VERSION.SDK_INT>=Build.VERSION_CODES.O &&

(getActivity().getPackageManager().hasSystemFeature(PackageManager.FEATURE_EMBEDDED)));
}
```

This would return `true` if you appear to be running on Android Things (or a similar Android-based IoT platform), `false` otherwise.

The `android.hardware.lowpan` system feature is undocumented. It would appear to be tied to LoWPAN, part of a specification for a low-power radio communications stack. This was added to Android Things in DP 6.1.

Considerations

Long-time Android developers are used to changes in available platforms, whether that was the introduction of Google TV in 2010, tablets in 2011, smartwatches in 2014, Android TV in 2015, and so on.

However, most of those platforms were incremental changes. Android Things is substantially different, primarily because *you* are in charge of the hardware. That was not the case for most people for most Android platforms prior to Android Things.

Writing an app for Android Things is moderately different, but still an incremental change. Building a *product* using Android Things is much different than is building an app as a product.

And, with those differences, come things that you need to consider.

General App or Dedicated App?

Putting aside the hardware for a moment... is the app that you are building for an Android Things device *only* going to run on that device? Or, will this app run on other Android-powered hardware? TV-centric Android devices (e.g., Android TV, Amazon Fire TV) are a fairly close match for Android Things, as both represent devices that usually are not mobile. But, perhaps your app is also relevant for phones and tablets, that you happen to also be repurposing for an Android Things device.

If your app is exclusive to your Thing, then you can simply write the app.

If your app is going to run elsewhere, you need to decide how best to architect that, to handle the differences in environments:

- Have one app, with a lot of branching or strategy class implementations, to handle the differences between environments
- Have one app but use separate product flavors per environment, where your strategy classes lie in the flavors, with one flavor per major environment (e.g., Android Things, TVs, mobile devices)
- Have a series of library modules implementing pieces of the functionality, with multiple app modules that “mix-and-match” those library modules to give you the right combination for a given environment

- Have one library with core app functionality, with one app per environment that leverages the library (for common code) and handles all the environment-specific work
- Something else entirely

Update Strategy

With ordinary Android apps, your distribution strategy is part of what you need to decide upon. Some developers settle for distributing their app on Google's Play Store and nowhere else. In that case, distributing updates mostly is just a matter of uploading the updates to the Play Store Developer Console, with the Play Store taking over from there. If you choose different distribution channels (e.g., Amazon AppStore for Android, F-Droid, YANDEX), each channel will have its own mechanism for informing users of updates. If you distribute the app yourself, as a downloaded APK from your server, you will need to let users know about updates, whether from within the app or by external channels (e.g., announcements on a blog or Twitter account).

However, beyond updating your app, you also have to consider updating *the OS itself*. With ordinary devices, that is handled by the device manufacturer. With Android Things, *you* are the device manufacturer. Presumably, at some point, there will be official Google guidance for how to distribute updates to the OS. However, you will also need to consider any hardware-specific drivers or other modifications that you have made the OS, beyond just your Android Things app.

With Android Things, from an official standpoint, app distribution is tied inextricably to OS updates, published in unison through the Android Things Console. Whether or not that will be appropriate for your Thing depends a lot on how you are distributing the Thing itself.

Being able to update your app and the underlying OS is extremely important, for one key reason... security.

Security

If your Thing will never offer remote control or communicate with the Internet in any way, security is not much of an issue.

But, considering that the term is "Internet of Things", presumably many Things will actually connect to the Internet.

Whole botnets, like Mirai, are powered by IoT devices, where the device manufacturers did a poor job of securing those devices, allowing hackers to take them over and co-opt them into attacking Web sites. Litigation in this area is increasing, particularly from government agencies. Hence, it is in your interest to aim to secure your Thing better than your predecessors have in the Internet of Things.

For example:

- If you assume that your Thing will only be used on a private network, and therefore will not be subject to external attacks... perhaps you should check this and warn the user if your Thing appears to be visible to the Internet.
- If you are offering direct remote control, such as an on-device Web app and Web service that users can work with from desktop browsers or mobile apps, take steps to prevent outsiders from being able to work with that Web technology. An Android Thing usually will serve a small, countable audience, not countless millions or billions of people. So, for example, in a Web app or Web service, consider adding a random value in a path segment of URLs, so that the URLs cannot be guessed by those not configured to use the Web app.
- If your Thing supports WiFi, do not assume that the encryption is particularly good. In particular, whoever set up the WiFi network may have chosen a weak passphrase to secure the WiFi, and some WiFi networks are completely open. Encrypt your own data in motion, such as via SSL.

Device Catalog: BlackBerry

BlackBerry — formerly Research In Motion — has been a long-standing player in mobile devices. Their BlackBerry two-way pagers and early smartphones help set the stage for Android, iOS, and those that followed.

BlackBerry and Android have had an interesting history.

In 2011, BlackBerry leapt into the tablet arena with the Playbook, and the 2.0 version of the Playbook OS supported running carefully repackaged Android applications.

While the Playbook itself had modest success, the ability to distribute Android applications to BlackBerry devices continued with their BlackBerry 10 (BB10) platform, where they offered several phones that could run Android apps. Originally, these had to be specially packaged for BB10, and that is still a common course today. However, in concert with offering the Amazon AppStore for Android on BB10, BlackBerry made it possible to install ordinary APK files as well. Many developers have enjoyed success distributing their app through BlackBerry World (the primary distribution channel for apps to BlackBerry products) and Amazon Appstore for Android.

In 2015, BlackBerry continued their Android push with the BlackBerry Priv, a device designed from the outset to run Android. The Priv comes with a full suite of BlackBerry-related software, including the legendary BlackBerry Messenger (BBM). However, much of that software — including BBM itself — is available on the Play Store for ordinary Android devices. In general, from the standpoint of an Android app developer, the Priv is no different than an Android device from any other major manufacturer. The Priv is even part of the Google Play ecosystem and comes with the Play Store and Google Play Services.

Putting the Priv aside, though, getting your app going on BB10 is a bit more of an adventure. This chapter will describe a bit about what is involved in getting your Android app to BB10 devices.

I Thought BlackBerry Had Their Own OS?

They do.

However, current versions of that OS — this chapter was last updated when version 10.3 was the latest shipping version — contain an Android runtime environment. BlackBerry OS can run Android apps alongside apps written natively for BlackBerry OS or running on other runtimes (e.g., Adobe AIR). This gives developers a wide range of ways to get their app onto modern BlackBerry devices. However, it does mean that our apps may have somewhat less direct access to hardware, as there is another layer between us and that hardware.

What Else Is Different?

At its core, BlackBerry is a device manufacturer, no different than any other manufacturer that you may have dealt with previously. The biggest difference is BlackBerry's ability to run Android applications that you prepare for their devices.

That being said, the world of BlackBerry is a *bit* different than what you may be used to.

Hardware

BlackBerry makes phones with a variety of capabilities, much as do other manufacturers. You should be writing your apps to support a range of device characteristics, such as screen size and density. That will help you with BlackBerry support, just as it helps you with support for other manufacturers' devices.

Note, though, that BlackBerry has some history which will affect their device designs, and your apps by extension. Notably, BlackBerry has been renowned for their hardware keyboards. While not all BlackBerry devices today have such keyboards, it is likely that BlackBerry will ship keyboard-equipped devices for some time to come.

This has three impacts upon you as a developer:

1. Do not assume that the user is using a soft keyboard. Usually, this is not a problem from a programming standpoint, though you may wish to take it into account in documentation.
2. Do not assume that the user always uses the touchscreen to navigate. Some BlackBerry users may use the hardware keyboard for navigation. This is particularly true for users who gravitate towards hardware keyboards for accessibility reasons. Your app should support proper focus to allow it to be navigated without accessing the touchscreen, to the greatest extent possible.
3. BlackBerry keyboards often have not been "slider" keyboards (i.e., ones that might slide under the display when not in use). Rather, they are always available, below the screen. This will often result in somewhat smaller screen sizes, and odd aspect ratios, compared to what you are used to. There simply is not enough room for a large touchscreen *and* an always-available physical keyboard without having an excessively large device. The BlackBerry Q10, for example, has a 720x720 resolution screen, and most Android developers do not encounter square screen resolutions. You will need to take this into account, but only in cases where such an aspect ratio might cause you problems (e.g., full-screen image backgrounds).

BlackBerry OS 10.3

Beyond the hardware, the BlackBerry OS — and the Android runtime environment that executes our Android apps — puts some limits on what we can do in our apps. Of note:

- BlackBerry OS 10.3's Android environment uses Android 4.3 (API Level 18), so apps that have an `android:minSdkVersion` higher than that will not be eligible to run on 10.2 devices.
- Some classes will be non-functional, particularly those related to telephony capabilities. You cannot use `SmsManager`, for example.
- Some types of apps will not work well, because the Android runtime runs alongside other non-Android apps on the hardware. Replacement home screen implementations, for example, are unlikely to work. Similarly, background apps that display a UI (e.g., Facebook "chatheads"-style popups) using `SYSTEM_ALERT_WINDOW` are unlikely to work.
- App widgets are not supported. So long as the app widget is a non-essential feature of your app, this should not be a problem — after all, the user does not *have* to use your app widget on any Android device. However, if the *sole purpose* of your app is to provide an app widget, such an app will not be useful on BlackBerry.

- Not all connectivity is surfaced in the Android runtime from the underlying hardware. For example, WiFiDirect and direct USB access are all unavailable.

Navigation

Like most Android tablets, BlackBerry devices offer little in the way of physical or off-screen navigation buttons. For example, there is no BACK button. However, a navigation bar will contain a BACK soft button for users. If your app takes over the full screen, this bar will not be there all the time, but a swipe down from the top of the screen should expose it.

Similarly, your menu will not be accessed via a MENU key, but rather via a downward swipe to expose the menu. This also means that any special MENU-button logic of yours may not work, if you are using the MENU button for things other than displaying the action bar overflow or other form of options menu.

Nothing Googly

As with other devices cited in this book, the BlackBerry series of devices lack support for Google's proprietary apps. For app developers, this means that you lack access to Google Play Services and the various APIs exposed by it, such as Maps V2 and Google Cloud Messaging.

BlackBerry does offer its replacement for GCM, in the form of the BlackBerry Push Service. However, for maps, they steer you towards using `geo:` `Intent` structure and `startActivity()`.

If you are dependent upon other APIs offered by Google Play Services (e.g., `LocationClient`), you will need to reconsider your use of those APIs if you wish to ship on devices that are outside the Google ecosystem.

Package Name Length

The BlackBerry Android runtime appears to only support package names of 29 characters or less. The build tools will fail if your package name is longer than 29 characters.

Note that this should only pertain to the "application ID" role of a package name, not the "hey, where does `R.java` get generated?" role of the package name. Hence, it should be possible to replace the application ID of your app via a product flavor, to

give yourself a shorter identifier while not breaking your source code references to resources.

What Are We Making?

This might seem like an odd question. After all, the point of this chapter is to make an Android application that can run on BlackBerry devices.

Up until early 2014, that meant you had one option: write a BAR.

A BAR (BlackBerry ARchive, presumably) is a repackaged version of an Android APK, designed to be distributed by BlackBerry and installed on BlackBerry devices. Most of the tooling supplied by BlackBerry surrounds this process of validating that an APK should abide by BlackBerry's requirements and converting that APK into a BAR.

Starting with BlackBerry OS 10.2.1, though, BlackBerry device users can download and install an APK directly. That APK still needs to work within the confines of the BlackBerry Android runtime and must avoid things that will not work there (e.g., replacement home screens). But the user is no longer reliant upon the developer going ahead and creating a BAR file.

So, what you are creating depends a bit on how you want to distribute the app:

- If you wish to distribute via BlackBerry world, you will need to repackage your APK as a BAR file
- If you wish to distribute through Amazon Appstore for Android, or if you wish to self-distribute, you can still ship an APK, though you may wish to use some of BlackBerry's tools to help ensure that your APK will be compatible

Getting Your Development Environment Established

By and large, developing an app to run on the BlackBerry OS Android runtime is the same as is developing an app to run on any other Android environment. You have your standard choices of IDEs and other development tools, the ability to use third-party libraries, and so forth.

Where things start to differ is in testing, where BlackBerry OS ships a Simulator that fills a role similar to that of the Android emulator. To use the Simulator, you will need to package your app into a BAR file, and BlackBerry provides tools to assist you in that process as well.

Checking and Repackaging Your App

There are two basic technical steps for preparing your app for distribution through the BlackBerry World market.

The first is to validate that your app does indeed stick to APIs that are supported by the BlackBerry Android runtime. This helps prevent apps from appearing on BlackBerry World that are guaranteed to fail.

The second is to convert the APK into a BAR file. BAR files are used for all of the BlackBerry OS runtimes, including ones like Adobe AIR. Your BAR will contain the same stuff that is in your APK, plus some additional metadata, in a format shared by all of the other runtimes, for interpretation and use by the BlackBerry OS. Again, if you are not planning on distributing through BlackBerry world, you will not need to worry about a BAR file.

Android Studio Plugin

BlackBerry is distributing an Android Studio plugin to help with preparing Android apps for BlackBerry. Note, though, that this is an *Android Studio* plugin, not a Gradle plugin. On the plus side, this gives you new BlackBerry-related options in the Android Studio UI. However, since Android Studio evolves frequently, there is a decent chance that the BlackBerry plugin will not work on some newer Android Studio builds, until BlackBerry catches up.

This plugin includes:

- an `adb` proxy that allows Android Studio to work with BlackBerry devices and running simulators
- a UI for obtaining a “device debug token”, required to allow you to test your apps on BlackBerry devices
- package an APK as a BAR

Standalone GUIs

You also have the option for doing BlackBerry-related chores from standalone GUIs, independent of any IDE.

Basically, both IDE plugins simply provide menu options and toolbar buttons for launching the standalone GUIs from within the IDE. If you are not using Android Studio, those GUIs are available independently that you can run like any other desktop development tool.

BlackBerry 10 Simulator

BlackBerry distributes VMWare images that embody a BlackBerry 10 Simulator. You can use these with VMWare Player (Windows and Linux) or VMWare Fusion (macOS), versions 3.1 or higher.

The Simulator fills a role similar to that of the standard Android emulator, allowing you to test your apps for BlackBerry OS without necessarily having a Blackberry OS 10 device, or for testing scenarios that are difficult to test with actual hardware.

In particular, the Simulator offers a wide range of simulated input and output, including:

- Simulated sensor input for the accelerometer and ambient light sensor
- Using a development machine's Bluetooth adapter for testing Bluetooth
- Simulated NFC tags

In addition, the Simulator supports some of the same types of simulated input that you see with the Android emulator, such as simulated GPS fixes and simulated incoming phone calls.

The VMWare image will have its own IP address, which you can obtain from the Simulator running in the image. You can then deploy your BAR to it using the `adb` proxy, which you can launch from your IDE or the standalone GUI.

Developing on Hardware

A BlackBerry OS 10 device can run either signed or unsigned BAR files. Unsigned BAR files, though, require a one-time upload of a "debug token", the creation of which requires the same credentials as you would use to sign the BAR in the first place.

How Does Distribution Work?

As with any environment where the Play Store is not available, developers have to determine how best to get their apps to the users of BlackBerry devices.

As with most non-Play Store ecosystems, there is the official solution... and then there are the other solutions.

BlackBerry World

BlackBerry's own "market" is BlackBerry World. This supports native BlackBerry apps, plus those for runtimes like the Android runtime. So long as your app is packaged as a BAR, it should be able to be released through BlackBerry World, much like how you distribute an APK through the Play Store.

BlackBerry World is a curated marketplace, meaning that BlackBerry staff will review your app submissions and may reject them if they violate requirements or other terms.

Beyond that, BlackBerry World has most of the standard capabilities that you would expect from an app market, such as paid apps, in-app purchases, multiple billing options (PayPal and carrier billing), control for distribution to various countries and mobile carriers, etc.

Amazon Appstore for Android

However, many Android developers will probably elect to distribute through Amazon Appstore for Android, now that it ships on new BlackBerry devices.

Partly, it is for improved reach: by distributing through Amazon's channel, you reach the Kindle Fire and Fire TV series of devices, plus any other Android devices that happen to have the Amazon Appstore for Android installed.

Partly, you can distribute in the form of an APK, rather than having to mess around with registering for BlackBerry World, creating BAR files, and the like.

Alternatives

Starting with BlackBerry OS 10.2.1, you can distribute APK files through your Web site or similar means, and BlackBerry users can download and install them.

However, nothing is done to validate that the apps you download will work on BlackBerry OS's Android runtime, as they may use APIs that are not available.

Also note that the BlackBerry OS settings have an equivalent to the Android "allow apps from other sources" setting that must be enabled for such installation to occur.

As an example, you can install the F-Droid "market" app to download free and open source Android apps to a BlackBerry OS device. How many of the apps distributed through F-Droid will work on BlackBerry OS, besides F-Droid itself, is unknown.

Device Catalog: Android TV

Google not only offers the Chromecast, but also offers Android TV as a way to get content "into the living room". Android TV devices — whether they be set-top boxes or are integrated into televisions directly — run Android apps directly, unlike Chromecast. Android TV, therefore, is a competitor to devices like Amazon's Fire TV.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having read the chapter on "ten-foot" user experiences is also a good idea.

Hey, Wait a Minute… I Thought the Name Was "Google TV"?

You can be forgiven for any confusion over the names.

Google TV was Google's initial attempt to get content onto televisions. Debuting in 2011, Google TV has some of the same characteristics as does Android TV:

- It ran Android apps directly (at the time, on Android 3.1)
- It could be a dedicated set-top box, integrated into other sorts of media players (e.g., Blu-Ray), or integrated into televisions

However, Google TV did not prove all that popular.

In 2014, Google announced that they were no longer supporting app development for Google TV, much to the consternation of the ~17 people still using Google TV devices.

That being said, designing an app for Android TV resembles designing an app for Google TV or for any other "ten-foot" user experience. Hence, design guidance that you may run across for Google TV may have some tips that are still relevant for Android TV and other TV-centric Android environments.

Some Android TV Hardware

Android TV debuted in 2014. However, as of the end of 2014, there were a total of two Android TV device models... both available from Google. While other manufacturers had announced plans regarding Android TV, none were available at the time of this writing.

ADT-1

When Google TV was announced, there was a similar lack of hardware. To help ensure that there were TV-capable apps at the time of a wider Google TV rollout, Google offered free developer devices to various firms and solo developers.

Google did the same thing with Android TV, where they offered a free ADT-1 device to qualified registrants. While deliveries of the ADT-1 were spread out over a few months, they generally arrived significantly in advance of production Android TV hardware.

Figure 1119: ADT-1 Developer Android TV Device

Nexus Player

The first production-grade Android TV device is Google's own Nexus Player. As with the ADT-1, the Nexus Player has an HDMI port for connecting to a TV (or projector, monitor, etc.). It also has its own remote control and power adapter.

Figure 1120: Nexus Player Android TV Device

Both the ADT-1 and the Nexus Player are presently running Android 5.0.

What Features and Configurations Does It Use?

Android has built into the SDK a fair bit of device flexibility. Most of this comes in the form of configurations (things that affect resources) and features (other stuff). If your application can handle a range of configurations and features, or can advertise that they need certain configurations or features, they can handle Android TV or arrange to not be available for Android TV on the Play Store.

Screen Size and Density

Android TV devices are always categorized as `xlarge` screen size.

Densities, however, are a bit more complicated.

Android TV is for use with HDTV, whether Android TV is integrated into the television or it comes as an external set-top box. There are two predominant HDTV resolutions, known as 720p (1280x720) and 1080p (1920x1080). A 1080p television will

be categorized as an `xhdpi` density device. A 720p television will be categorized as a `tvdpi` device. `tvdpi` is for devices around 213dpi, in between `mdpi` and `hdpi`. In practice, you might elect to skip `tvdpi` for your drawable resources, allowing Android to resample your `mdpi`, `hdpi`, or `xhdpi` drawables as needed.

Input Devices

Android TV will not normally be navigated using a touchscreen. Instead, the normal form of input will be a D-pad remote. Developers will need to ensure that their apps are navigable this way.

Other Hardware

Android TV has no sensors, no camera, no microphone, and no telephony features. As such, any application requiring such features will not run on Android TV and will not even show up in the Play Store for such devices.

Bear in mind that some of these will be driven by permissions. If you ask for the `SEND_SMS` permission, Android will assume you need android.hardware.telephony unless you specifically state otherwise, via a `<uses-feature>` element for `android.hardware.telephony` with `android:required="false"`.

What Is Really Different?

Beyond the features and configurations, there are other things about Android TV that will depart from what you might expect for an Android environment, due to the nature of the TV set-top box platform and the Android implementation upon it.

Overscan

Since Android TV typically uses a television for its primary output, overscan can be an issue. Addressing overscan is covered as part of the chapter on the “10 foot UI”.

Ethernet

While Android TV devices will generally be connected to the Internet, it may not be via WiFi. Since Android TV devices generally are not portable, some will have Ethernet jacks, and hence some users will elect to wire in their Android TV as opposed to using WiFi.

The upshot is that you should not assume that `WifiManager` will necessarily give you useful results. Also, `ConnectivityManager` should report wired Ethernet as `TYPE_ETHERNET`, added in API Level 13, when you call methods like `getActiveNetworkInfo()`.

Location

Generally speaking, Android TV devices will tend not to move, earthquakes and large dogs notwithstanding.

As such, Android TV devices do not have GPS receivers. Rather, location is determined in an approximate fashion via address-based lookups, using a postal code. Hence, asking Android for a GPS fix on a Android TV device will be ineffective.

However, since users of Android TV devices tend not to be moving much at the time, it is a bit more likely than normal that they will want information about some location other than where they are. If your app is exclusively tied to providing information about their current location, you may wish to consider how you could extend your app to help users get information about other places that they may be interested in.

Media Keys

Android TV devices are usually manipulated by remote controls. Some of these remotes will have lots of buttons, such as media-specific buttons for play, pause, etc.

The `KeyEvent` class has had support for some media buttons since API Level 3, mostly for use with wired headsets. API Level 11 added a bunch more media buttons. Your Android TV application may wish to respond to these, via `onKeyDown()` in a `View` or `Activity`.

In particular, an Android TV application should not be using on-screen controls for play, pause, etc., as they take up screen space that probably could be put to better use. Rather, use layouts that offer such controls for touchscreen devices (e.g., phones and tablets) but rely on the media buttons for non-touchscreen devices.

Getting Your Development Environment Established

Android TV emulator images are available in the SDK Manager for Android 5.0 and above:

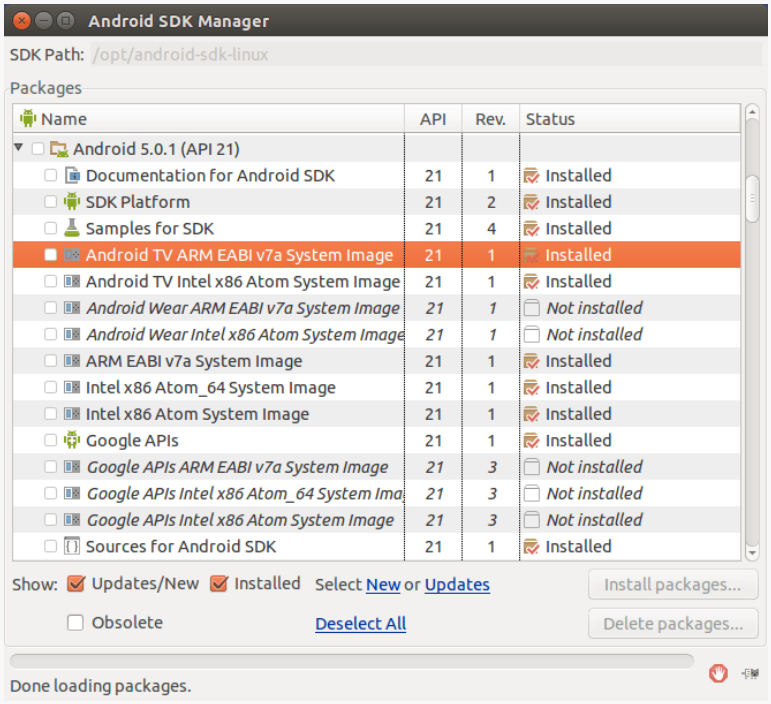


Figure 1121: Android TV Emulator Images in SDK Manager

Your AVD Manager can then help you set up TV emulator images, for rather large theoretical screen sizes:

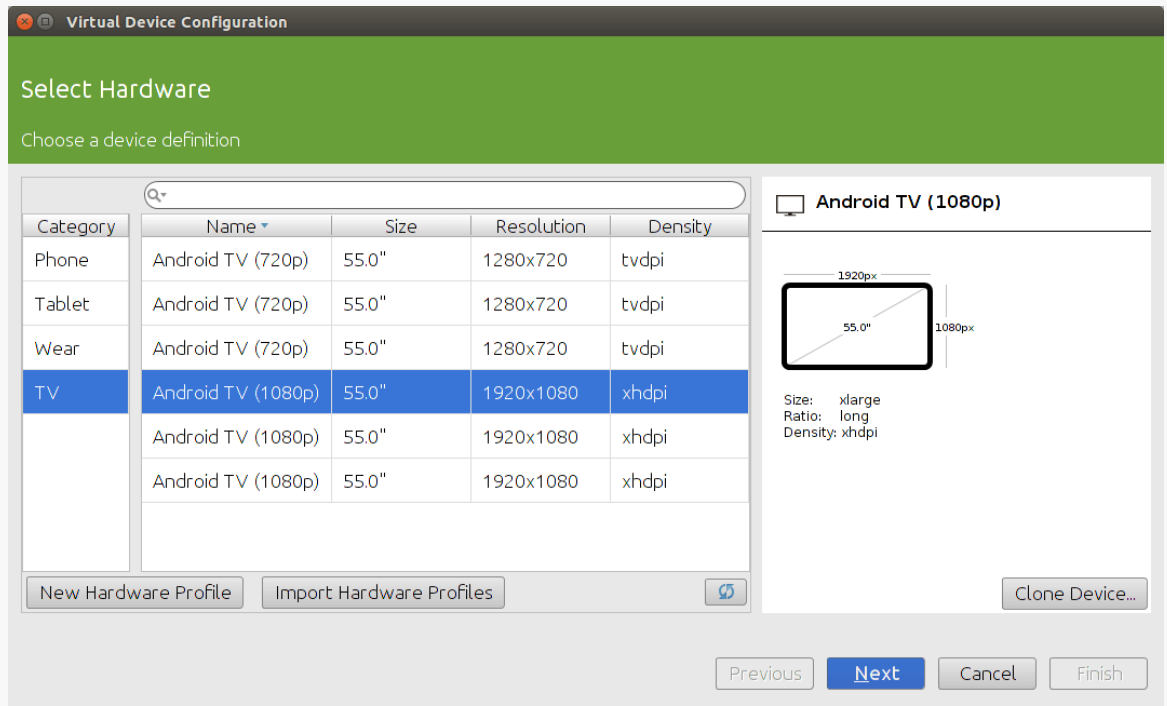


Figure 1122: Android TV Hardware Profiles in AVD Manager

The rest of the emulator setup is the same as you would have for phone or tablet emulators.

The emulator even has “leanback” UI characteristics, rather than a standard Android emulator home screen:

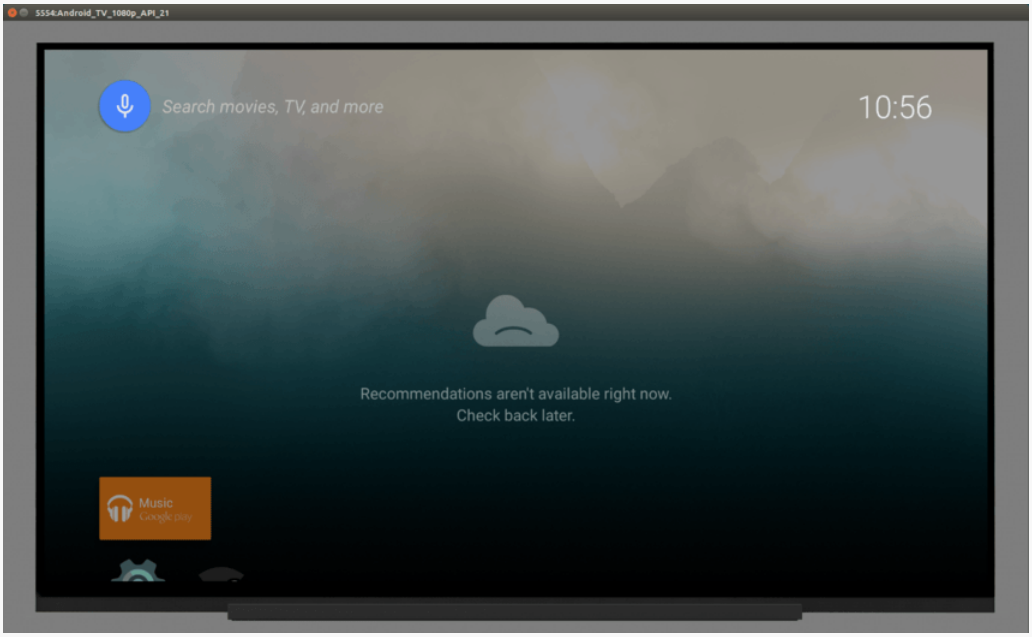


Figure 1123: Android TV 1080p Emulator, As Initially Launched

Connecting to Physical Devices

The ADT-1 and Nexus Player have micro-USB ports for debugging purposes. This, though, requires that your player and your development machine be within USB cable reach of one another. The author of this book uses a pico projector to be able to work with TV-native devices (Android TV, Fire TV, etc.) or external displays (HDMI, MHL, etc.).

To enable an Android TV for debugging, you will need to enable developer options, much like you do for a mobile device (click on the build number 7 times in the Setting's About screen). Then, in the "Developer Options" screen, you can go into Debugging and elect to enable "USB debugging".

How Does Distribution Work?

Your app probably falls in one of three buckets: you want it on Android TV (along with other devices), it *only* supports Android TV, or it will not work on Android TV. Whichever of those buckets best fits your device will determine the manifest settings you will want to ensure that the Play Store (and perhaps other third-party markets in the future) will honor your request.

Getting Your App on Android TV

The first criterion for getting your app visible to Android TV devices on the Play Store is to add a `<uses-feature>` element to your manifest, indicating that you do not require the `android.hardware.touchscreen` feature:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="false"/>
```

By default, Android assumes that you need a touchscreen, and so without this clarification in your manifest, you will not appear in the Play Store.

Also, add similar `<uses-feature>` elements for any hardware that you might like to use where available but do not absolutely need, particularly hardware that Android TV may lack. The documentation outlines the features that Android TV devices will likely lack.

You need to have an activity that has an `<intent-filter>` for `ACTION_MAIN` and `CATEGORY_LEANBACK_LAUNCHER`. This will be your launcher activity on Android TV devices, instead of your `ACTION_MAIN`/`CATEGORY_HOME` activity. Of course, you are welcome to have one activity serving as the launcher for both types of devices, if you can come up with one with a solid presentation for both mobile devices and TVs.

In addition, do not have any activities with `android:screenOrientation` set to `portrait`, as Android TV devices always display in landscape

Supporting Only Android TV

If your app only supports Android TV, in addition to the above requirements, you should also add one more `<uses-feature>` element to your manifest:

```
<uses-feature android:name="android.hardware.type.television"
android:required="true"/>
```

This will filter you out of the Market for all non-TV environments.

Avoiding Android TV

If your app specifically is untested on Android TV, you need to have something in the manifest that will keep you *off* Android TV devices' views of the Play Store. The easiest is to say that you need a touchscreen:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="true"/>
```

Note that `true` is the default setting for this particular feature, though putting it in your manifest to remind you that you do require a touchscreen is a good idea.

Device Catalog: Amazon Fire TV and Fire TV Stick

Amazon has joined the TV set-top box fray with the Fire TV and, more recently, the Fire TV Stick. These devices are powered by FireOS, Amazon's variation of Android. And, as with other Amazon FireOS devices, like the Kindle Fire tablet series, you can write apps that run on Fire TV and the Fire TV Stick.

This chapter will review these devices from a developer's standpoint, to help you create apps for this platform.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Reading the chapter on the ten-foot user interface is also recommended, either before or after this chapter.

Also, some sections will make reference to Android TV as a point of reference.

Introducing the Fire TV Devices

As of August 2017, there were two major flavors of Fire TV device: the original Fire TV, and the Fire TV Stick. Both run FireOS, Amazon's name for their derivative of Android.

Fire TV

As with most Android-powered set-top boxes, the original Fire TV is small and is designed to connect with your television (or monitor, or projector, or whatever) via HDMI:

Figure 1124: Fire TV

It comes with a small wireless remote designed for basic controls, akin to what you might find on other streaming boxes or similar entertainment devices:

Figure 1125: Fire TV Remote

Optionally, you can get a gaming controller that works with the Fire TV. While the regular controller is fine for navigating the Fire TV UI, the gaming controller will be more suitable for more serious game play:

Figure 1126: Fire TV Gaming Controller

While the Fire TV is powered by Android, the on-device UI is definitely targeting a set-top box environment. The home screen is dominated by media, coming from whatever supported streaming content providers you have set up with the Fire TV (e.g., Amazon Prime):

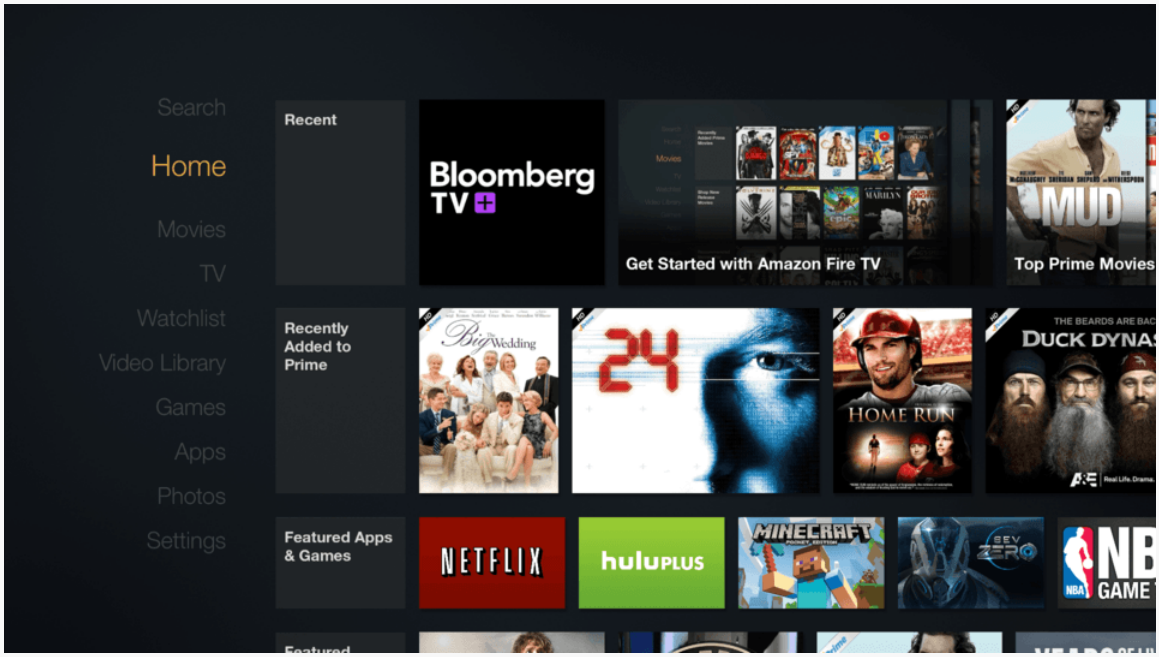


Figure 1127: Fire TV Home Screen

The "Apps" section shows a mix of what is installed and what is available for you to download, with "cloud" icons indicating apps that are available but are not presently installed:

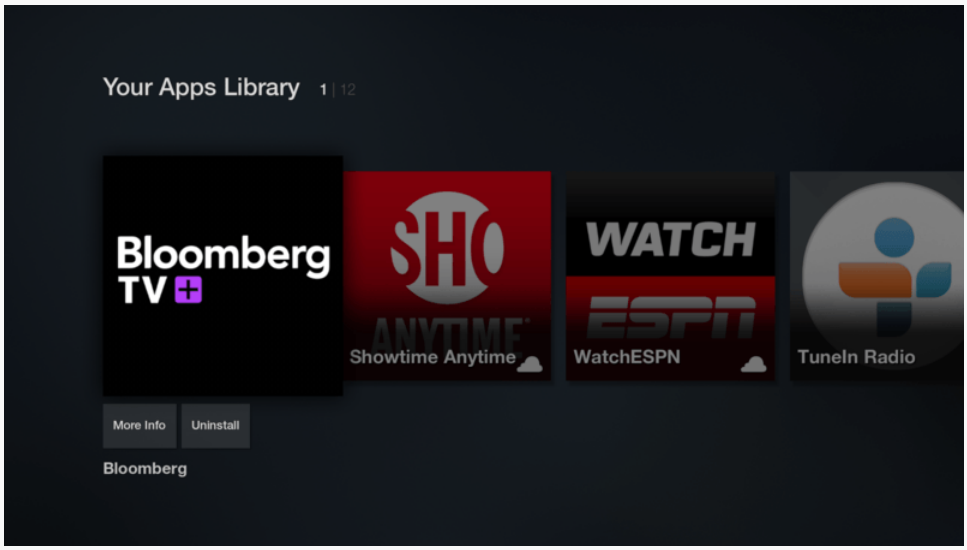


Figure 1128: Fire TV "Your Apps Library"

FIre TV Stick

The Fire TV Stick is physically substantially smaller than is the Fire TV, designed to more closely resemble the Chromecast:

Figure 1129: Fire TV Stick, with Remote, AC Adapter, and HDMI Extension Cable

The user experience of a Fire TV Stick, though, is largely the same as with a regular Fire TV. Both run the same FireOS environment, with the same sort of browsing metaphor.

The biggest difference is that the Fire TV Stick is less powerful (less RAM, weaker CPU and GPU). For conventional apps, this is unlikely to be a problem, as the Fire TV Stick is as powerful as many standard Android phones and tablets. Games, however, are more likely to stress the hardware.

From the user's standpoint, the Fire TV Stick's big selling point is its low price, about a third of what the Fire TV costs.

What Features and Configurations Do They Use?

The Fire TV device family behaves a bit like a "mashup" of other TV-centric devices (Android TV, OUYA, etc.) and the Kindle Fire series of tablets.

Screen Size, Density, and Orientation

One area where the Fire TV series differs from pretty much anything that came before it comes with the behavior of the screen. As with other TV-centric devices like Android TV, the assumption is that the screen is locked to landscape. However, beyond that, Amazon has struck out on its own in terms of screen characteristics.

With Android TV, the screen size and density are based on the type of display that the Android TV box is plugged into.

With the Fire TV devices, your code deals with a rendering surface of 1920x1080 pixels, regardless of whether the device is plugged into a 1080p screen or something else (720p, 4K, etc.). The density is always treated as `-xhdpi`, giving you a resolution of 960dp by 540dp, and a `-large` screen size. The hardware will handle scaling the output down (or, in theory, up for 4K) as needed.

On the plus side, this simplifies app development, as you do not need to deal with different screen capabilities yourself. However, it remains to be seen how well the Fire TV will scale the output.

Also, note that Fire TV devices do not address overscan when they apply this scaling. Hence, you will want to keep your content away from the edges of the available space, only showing your background there, in case those edges are not visible on some televisions.

As with any TV-centric device, since the screen is locked to landscape, activities that are locked to portrait (e.g., `android:screenOrientation="portrait"` in the manifest for the activity) will behave oddly. If you are targeting devices like the Fire TV series, be sure to allow your activities to work in portrait *or* landscape, not just portrait.

Input Devices

As noted earlier, Fire TV devices ship with a simple remote control, designed for media management (play, pause, etc.) and basic D-pad navigation. Some users may also elect to pick up a Fire TV game controller or use other Bluetooth game

controllers. Technical details for working with these input devices can be found later in this chapter.

Conversely, Fire TV devices do not have touchscreens. You cannot assume that the user can tap on random portions of the screen. Instead, you need to make sure that your app is completely reachable via a D-pad. This is described in greater detail in the chapter on the "ten-foot UI".

Hardware Features

As a set-top box, Fire TV devices lack a lot of hardware that you normally associate with phones and tablets, such as:

- GPS
- camera
- microphone
- sensors
- telephony

The microphone is actually a bit complicated. The Fire TV remote has a microphone and a button to activate it. The Fire TV Stick remote does not have a microphone. There is a Fire TV Remote app available on the Play Store and Amazon Appstore for Android that offers microphone input as well. However, all these microphones are just for Fire TV voice search. There is no means for developers to use these for arbitrary audio input, the way that a microphone on a phone or a tablet can.

What Is Really Different?

Some things are "really different" simply because the output screen is TV-sized, not phone-sized or tablet-sized. See the chapter on the "ten-foot UI" for more about this.

Some things are "really different" in the same way that they are "really different" for the Kindle Fire series, such as having nothing "Googly", like Maps V2 or anything else from Play Services.

The biggest additional "really different" item is the limitation on where apps can come from, which is discussed later in this chapter as part of the overall app distribution options for Fire TV devices.

Note that Fire TV devices have their own way of representing notifications. A standard Android `Notification` will not appear anywhere on a Fire TV device. Instead, you will need to support the Fire TV series' own notifications API. While their API is modeled after Android's `Notification` and `NotificationManager`, they do have their own classes, and the results are more reminiscent of `Toasts` and `Dialogs`.

Also note that Fire TV devices do not display an Android action bar, representing those items in a pop-up menu triggered by the MENU button of the Fire TV remote control.

Casting and Fire TV

The success of Chromecast means that TV-connected Android devices will tend to be compared to Chromecast, particularly in terms of how apps on phones and tablets can work with the TV-connected device.

At the time of this writing, Amazon has not shipped anything for Fire TV that would enable apps using `MediaRouteActionProvider` and `RemotePlaybackClient` to work with Fire TV. In principle, Amazon or somebody else could implement a `MediaRouteProvider` and distribute that as an app that goes on ordinary Android phones and tablets, where that `MediaRouteProvider` enables media routes pointing to a connected Fire TV. The `MediaRouteProvider` would forward requests from the phone or tablet to the Fire TV, which would be displayed by some app on the Fire TV.

Amazon more formally supports the "Discovery and Launch" (DIAL) protocol with Fire TV. DIAL allows an app on a phone or tablet to launch apps on Fire TV. Unfortunately, documentation for this is lacking at the present time.

Also, the Fire TV series can serve as a Miracast endpoint. In Settings > Display & Sounds, you will find an option to "enable display mirroring". This turns on Miracast support, allowing mobile devices to mirror their screen content via the Fire TV to the connected television, projector, or monitor. This also works with Android's `Presentation` support, allowing you to direct separate content to the Fire TV's "mirror" than what you show on the device's own screen.

Getting Your Development Environment Established

Developing for the Fire TV series, as with developing for the Kindle Fire series, is accomplished using an actual Fire TV or Fire TV Stick device.

The primary thing notably different about testing your app on a Fire TV device is that it does not use a micro USB cable for the `adb` connection, the way that you may be used to from testing with most Android hardware. If you can find a suitable USB cable to bridge between your development machine and the Fire TV, reportedly you can use USB debugging. Otherwise, you can use `adb` over the network.

To set this up, you must first enable USB debugging, much like you do with other Android devices. On a Fire TV device, this is in Settings > System > Developer Options:

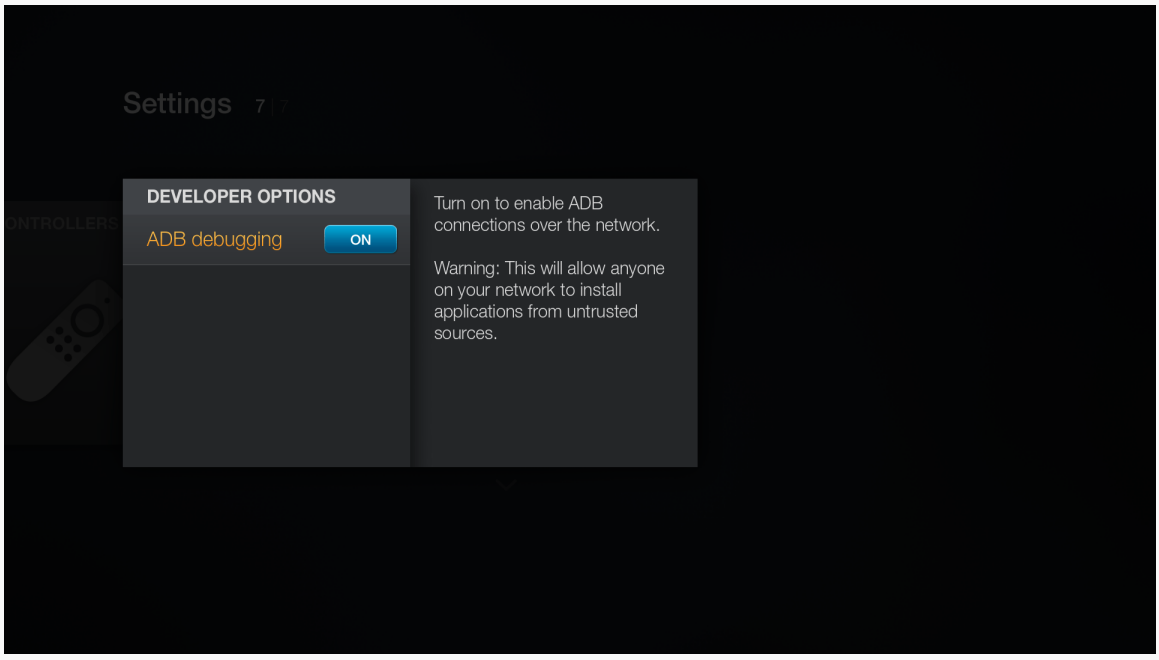


Figure 1130: Fire TV Developer Settings

You will then need the IP address of your Fire TV device. Most likely the easiest way for you to find that is via the Settings > System > About > Network screen:

Figure 1131: Fire TV Network Settings

You can then run `adb connect` (followed by the IP address) at a command prompt to make the connection. You may need to stop and restart `adb` before doing this, via `adb kill-server` and `adb start-server`.

At this point, if all is set up properly, `adb devices` will list the Fire TV device's IP address, and the Fire TV device will be accessible from your IDEs and other development tools.

Note that this `adb` setup will persist until `adb` is next restarted, even if the Fire TV device is powered down. You can use `adb disconnect` (followed by the IP address) to break the connection if you no longer need it.

Working with the Remote and Controller

Your Fire TV users will be interacting with their Fire TV using the supplied remote, the Fire TV gaming controller, or other controllers.

Wireless Remote

Mostly, the wireless remote offers buttons that you should be handling already, assuming you are implementing a ten-foot UI or are otherwise correctly handling focus management and accessibility. BACK and the D-pad buttons will work on the Fire TV much as you would expect.

The Fire TV device remote has a MENU button, which will bring up an old-style Android options menu. Your simple action bar items, particularly those in the overflow, will appear in this menu without issue. Most likely you will want to skip the action bar on your Fire TV apps, though, which means that action views, action providers, and the like will need to be replaced with alternatives.

The remote also offers play/pause, rewind, and fast-forward buttons that map to their corresponding Android `KeyEvent` types (e.g., `KEYCODE_MEDIA_PLAY_PAUSE`). You can watch for these events in `onKeyDown()` of your activity to be able to respond to them.

Note that you do not have the ability to intercept home or voice search button presses on the wireless remote. And, the Fire TV Stick's remote does not offer the voice search button.

Gaming Controller

The wireless remote's input options are purely buttons. The Fire TV series gaming controller, like most such controllers, are designed for more "analog" input, where the force you apply to a stick might trigger slightly different behavior in a game.

As such, the gaming controller support in Android and FireOS takes a two-tier approach, with primary and secondary ways of handling events. If your code consumes the primary event, the secondary event is not triggered. The idea is that fairly "vanilla" apps might pay attention to the secondary events, as they tend to be more in common with the events you might get from the wireless remote or non-Fire TV devices. But apps that are more game-centric might pay attention to the primary events instead.

For analog inputs — the left and right sticks and the D-pad — the primary event input is supplied as `MotionEvent` objects, which you can pick up in methods like `onGenericMotionEvent()` in a `View`. Secondary event input comes in the form of standard D-pad events. So, an ordinary app will automatically support the sticks and D-pad, simply by accepting D-pad input.

Some buttons also take the two-tier approach. The A button (bottom one in the button cluster on the upper-right side of the controller) can be picked up either as a `KEYCODE_BUTTON_A` `KeyEvent` (primary) or a `KEYCODE_DPAD_CENTER` `KeyEvent` (secondary). The B button (right one) maps to `KEYCODE_BUTTON_B` (primary) and `KEYCODE_BACK` (secondary).

Other buttons just fire simple `KeyEvents` (e.g., left shoulder is `KEYCODE_BUTTON_L1`), while the two triggers just use `MotionEvents` (`AXIS_BRAKE` on the left, `AXIS_GAS` on the right).

Amazon's Fire TV site has full details of the options and how to use them.

How Does Distribution Work?

Like the Kindle Fire, Fire TV devices lack the Play Store. If you want your app to be available to Fire TV device users, you will need to explore other ways of promoting and delivering the app.

The principal — and nearly exclusive — way to get apps onto a Fire TV device is by listing them in the Amazon AppStore for Android.

However, for ordinary users, that is the *only* option. Most Android devices — including the Kindle Fire series — allow the user to download apps from Web sites, if they have the appropriate option checked in Settings. The Fire TV lacks this setting, and therefore ordinary users cannot download an app to the Fire TV from third-party app stores.

"Sideloading" an app using `adb` works, though this is usually only viable for developers or serious power users. Such apps will not appear in the home screen launcher and must be launched instead via Settings > Applications.

Getting Help

Amazon maintains a set of documentation related to Fire TV device development, along with a set of forums for asking Amazon-specific development questions regarding the Fire TV device series or their various SDKs.

Device Catalog: Samsung DeX

In 2017, Samsung released the Galaxy S8 and S8+. They also released the Samsung DeX, a docking station for the S8/S8+ that not only provides power, an Ethernet jack, and USB ports, but it also has an HDMI port.

Figure 1132: Samsung DeX, Front Top Showing Device Connector

Figure 1133: Samsung DeX, Rear Showing Ports

Moreover, not only can that HDMI port be used the way it is on many other Android devices (screen mirroring, `Presentation`, etc.), but also in a "DeX" mode that simulates a freeform multiwindow experience. In this mode, the touchscreen is turned off, and the user navigates the windows using a keyboard and mouse (USB or Bluetooth), as with a traditional desktop OS.

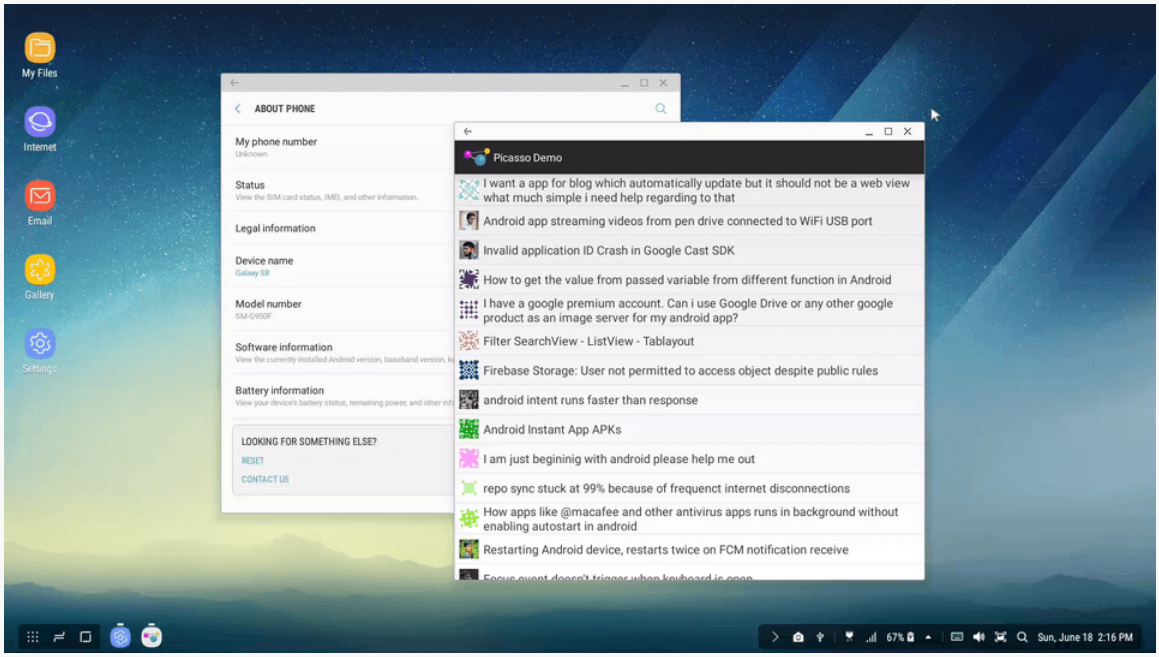


Figure 1134: DeX Mode, Showing Freeform-Style Windows

On the whole, developers do not seem to be concerning themselves too much with DeX — for example, as of mid-June 2017, there was exactly one question on Stack Overflow related to DeX. That being said, the DeX is an interesting demonstration of where Android 7.0's multi-window modes will head. Plus, it is yet another environment that puts keyboards and mice "front and center" for users and, by extension, app developers.

DeX Screen Modes

The user has two choices when docking their device in the DeX: screen mirroring mode and DeX mode.

Screen Mirroring

What Samsung describes as "screen mirroring" mode is pretty much what you would expect from an Android device connected to an HDMI display. By default, the contents of the touchscreen are mirrored on the HDMI display. And, if you use things like `Presentation`, you can display separate content on the HDMI display from what is shown on the touchscreen.

However, this mode may not be very popular, for one simple reason: the device is docked in the DeX in portrait mode. This means that the content shown on the HDMI device, by default, is in portrait mode. While you could lock your activity to landscape mode, so its `Presentation` appears in landscape, then the activity is in the wrong orientation on the touchscreen.

DeX Mode

More often than not, if people are bothering to put their devices in a DeX, it is to use Dex mode, with freeform-style windows.

This chapter refers to these windows as "freeform-style", as while the S8 and S8+ shipped with Android 7.0, they do not appear to have the freeform setting enabled. Hence, Samsung "rolled their own" freeform multiwindow implementation, and the developer (and user) experience in DeX mode will differ somewhat from what "official" freeform multiwindow mode is like.

Activities that are not resizeable will appear in portrait mode by default:

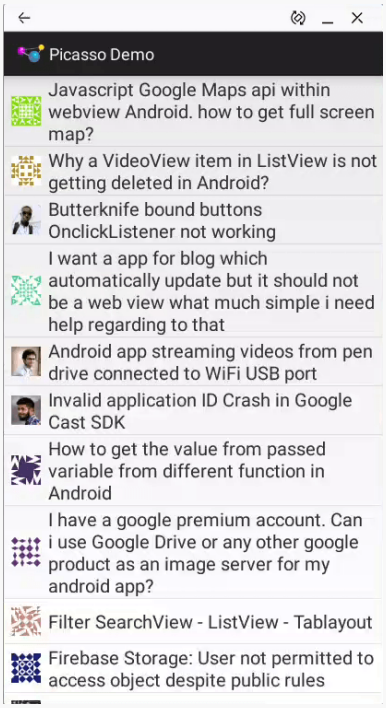


Figure 1135: Non-Resizeable Activity on DeX in Portrait

There is an icon in the title bar that allows the user to rotate the window to landscape:

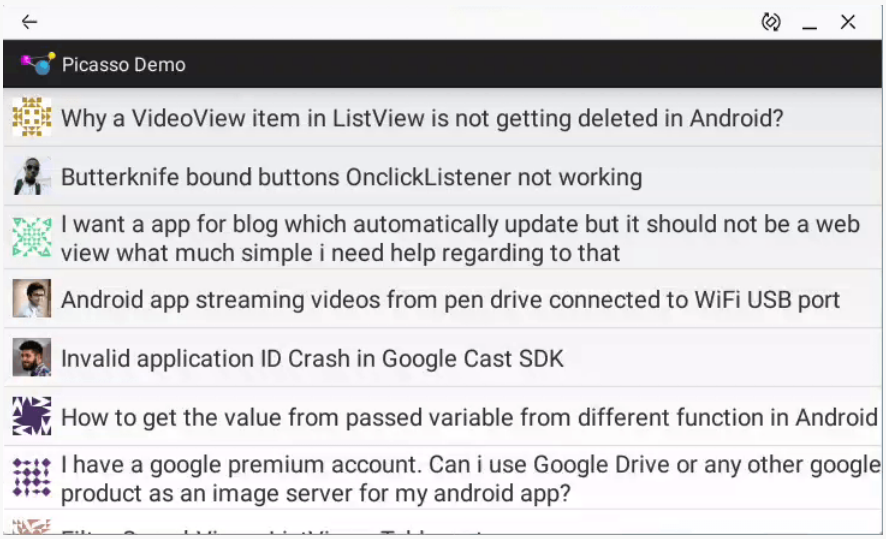


Figure 1136: Non-Resizeable Activity on DeX in Landscape

Activities that are resizeable — the `<activity>` or `<application>` explicitly has `android:resizeableActivity="true"` — can be resized by using a mouse and dragging the window edges, as with a traditional desktop operating system.

All windows, resizeable or not, can be minimized or maximized. Maximized windows will fill the screen.

Other App Impacts

While the screen, keyboard, and mouse are the primary changes that app developers will face when thinking about the DeX, there are other issues to take into account and opportunities to consider.

Configuration and State Changes

The strangest part of life in a DeX dock comes in the various state changes that your app can undergo. Sometimes, these are part of standard Android configuration changes. Sometimes, these are more distinctive to the DeX.

Moving In and Out of DeX

Here, we have many scenarios:

- A device is placed into the dock, and the device goes into DeX mode
- A device is placed into the dock, and the device goes into screen-mirroring mode
- A docked device is switched from DeX mode to screen-mirroring mode
- A device in DeX mode is removed from the dock
- A device in screen-mirroring mode is removed from the dock

(in theory, there should also be "a docked device is switched from screen-mirroring mode to DeX mode", but there does not appear to be an option for this)

The default behavior in all of these is:

- If your app is in the foreground at the time of the state change, your app will be killed and restarted
- If your app is in the background at the time of the state change, your app is killed and not restarted

Going into Dex mode will have your foreground app appear in the "system tray", which serves the same role in desktop mode as the overview screen does in standard Android. Clicking your app's icon will bring up your window... starting your app along the way.

The reason why Samsung does this by default is that switching to and from DeX mode is a fairly major change to your environment, specifically amounting to these configuration changes:

- `density`
- `orientation`
- `screenLayout`
- `screenSize`
- `smallestScreenSize`
- `uiMode`

In addition, there is a fairly substantial resolution change (2960x1440 to whatever the HDMI display supports).

If you can handle all of those changes without harming the user experience, add this `<meta-data>` element as a child of the `<application>` element in your manifest:

```
<meta-data android:name="com.samsung.android.keepalive.density" android:value="true"/>
```

With that element in place, when the user enters or exits DeX mode (e.g., puts the device into or removes it from the dock), your app is supposed to be left alone. In practice, this does not appear to work.

Networking

If the DeX dock is plugged into Ethernet, the device will switch to Ethernet connectivity when it is placed into the dock by default, though the user can configure this behavior.

If your app assumes that the Internet connection is via WiFi or mobile data, this is yet another scenario in which it might not be, to go along with Android TV, Android Things, and various miscellaneous Android devices offering Ethernet ports (e.g., Jide's Remix Mini PC).

Also, this means that your connectivity may change when the device is put into or removed from the dock... but, as noted above, your app may be killed anyway.

USB

In addition to working with keyboards and mice, the USB ports on the DeX dock are standard USB hosts. So, for example, the user can plug in a flash drive and browse that drive's contents.

In principle, the user can plug in any sort of USB device, and so long as the Samsung device has the appropriate drivers (or your app uses the USB APIs), the device should work.

Note that the USB ports are USB 2.0, not 3.0 or higher. Also note that they are classic Type A connectors, even though the power supply input comes from a Type C connector.

Debugging Interface

The DeX dock requires that you power it through a high-power USB port. A standard 500mA USB port will be insufficient. The problem is that most high-power

USB ports do not also support data, and in those situations you cannot use the USB cable for debugging.

Fortunately, what is often called "WiFi debugging" works, though technically it may be occurring over Ethernet, if the DeX dock has an Ethernet cable plugged in and the device is using it.

To use this style of debugging:

- With the Samsung device out of the DeX dock, plug it into your development machine as normal
- Run `adb tcpip 5555` from the command line, to turn on WiFi debugging
- Unplug the device from the USB cable and place it into the DeX dock
- Use Settings > About device > Status or some other means to identify the IP address of the device, bearing in mind that this might be either Ethernet or WiFi
- On your development machine, run `adb connect ...`, where `...` is the IP address of the device

You should get a message showing that you are connected, `adb devices` should show the connection, and Android Studio should be able to work with the Samsung device.

Use `adb disconnect ...`, where `...` is the IP address of the device, to drop this debugging connection.

Screenshots

When the docked device is in DeX mode, you cannot take screenshots through normal means (e.g., Android Studio, `adb shell screencap`). As with Android Things, you *can* record screencasts. And, given a short screencast, you can extract a frame to use as a screenshot, such as via the command-line `ffmpeg` utility.

However, the screencast (and any resulting screenshot) may be off, depending on the screen resolution of whatever HDMI display you have plugged the DeX dock into. The screencast will record at 2960x1440 resolution, which is the resolution of the touchscreen. However, it is very likely that your HDMI display cannot show that, and so the DeX dock will display at something like 1080p (1920x1080). The screencast will be at 2960x1440 resolution, with the external display's content shown in the upper-left corner, and black pixels filling the rest of the 2960x1440 frame.

For screenshots, you can crop the raw screenshot to get the portion that you need, dropping out the extraneous black bands:

```
adb shell screenrecord --verbose /sdcard/screen.mp4 --time-limit 1
adb pull /sdcard/screen.mp4
adb shell rm /sdcard/screen.mp4
ffmpeg -ss 0 -i screen.mp4 -t 1 -s 2960x1440 -f image2 temp.png
convert temp.png -crop 1920x1080+0+0 +repage result.png
```

Here, we use `adb`, `ffmpeg`, and ImageMagick's `convert` tools to record a one-second screencast, clip a 2960x1440 frame from it (to `temp.png`), then crop the upper-left 1920x1080 pixels to form the final screenshot (`result.png`).

For screencasts, there should be an equivalent way of having `ffmpeg` crop all of the frames to the desired resolution. The proof of this is left as an exercise for the reader.

Detecting DeX

Ideally, your app does not care whether it is running on a device in a DeX dock or not. In theory, if you do everything correctly in general (particularly multi-window support), then "it just works".

If you run into some DeX-specific quirk that you need to work around, you cannot use traditional `Build` values like `Build.PRODUCT`, because those reflect the *device*, not the dock.

Instead, Samsung apparently extended the `android.content.res.Configuration` class to have a DeX-specific field on DeX-capable devices:

```
private boolean iCanHazDeX() {
  boolean result=false;
  Configuration config=getResources().getConfiguration();

  try {
    Class clsConfig=config.getClass();

    if (clsConfig.getField("SEM_DESKTOP_MODE_ENABLED").getInt(clsConfig)
      ==clsConfig.getField("semDesktopModeEnabled").getInt(config)) {
      result=true;
    }
  }
  catch(Exception e) {
    // guess not!
  }
```

```
  return(result);
}
```

For More Information

Samsung has several pages devoted to developer documentation for the DeX, including:

- Instructions for modifying your app to be more DeX-friendly
- Information for testing your app without a DeX
- A FAQ

Trail: Appendices

Appendix A: CWAC Libraries

CommonsWare — the publisher of this book — has also published a series of open source libraries, collectively named the CommonsWare Android Components (CWAC). If you have read through the book, you will have seen many of these libraries.

This appendix lists all of the CWAC libraries. If the library is covered elsewhere in the book, the appendix links you to that coverage. Those that are not covered elsewhere will be described in this appendix, to accompany the online documentation found at the library's GitHub repository.

cwac-adapter

The cwac-adapter repository contains a small `AdapterWrapper` class that wraps a `ListAdapter`. The default implementation of all `AdapterWrapper` methods is to forward the request along to the wrapped `ListAdapter`. However, you can subclass `AdapterWrapper` to override that behavior.

cwac-colormixer

The cwac-colormixer repository holds a custom `ColorMixer View`, along with wrappers for using that `View` as a dialog, activity, or preference for use with `PreferenceScreen`.

`ColorMixerDialog` and `ColorPreference` are covered in the chapter on custom dialogs. The `ColorMixer` widget is similar to the implementation found in the chapter on custom views.

This library also contains a `ColorMixerActivity`, which you can use via `startActivityForResult()` to obtain a color, rather than by integrating the widget, dialog, or preference.

cwac-crossport

The cwac-crossport repository holds a version of a subset of the Design Support Library that has been "cross-ported" to work with `Theme.Material` and native activities, instead of `Theme.AppCompat` and `AppCompatActivity`. This is useful for projects that want to use some of those Material Design widgets but do not want the overhead of `appcompat-v7`.

cwac-document

The cwac-document repository holds a fork of `DocumentFile` that:

- Supports non-document `Uri` values
- Supports `Uri` values on older devices
- Offers convenience methods for working with the content (e.g., `copyTo()`, `copyFrom()`, `openInputStream()`, `openOutputStream()`)

cwac-layouts

The cwac-layouts repository contains a series of custom containers and related views.

The current contents of this library — `AspectLockedFrameLayout`, `MirroringFrameLayout`, and kin — are covered in the chapter on custom views.

cwac-merge

The cwac-merge repository contains `MergeAdapter`. It simply stitches together lots of smaller `ListAdapters` into one larger `ListAdapter` to put inside of a `ListView` or similar `AdapterView`. It also allows you to blend individual "row" `Views` with other `ListAdapters`.

One use of this is for section headers, using row `Views` for the headers and `ListAdapters` for the sections.

Another use is where you have multiple disparate data sources (e.g., queries across a few databases or `ContentProviders`), each with distinct row formatting, but you want to present them as one contiguous list.

cwac-netsecurity

The cwac-netsecurity repository contains a backport of Android 7.0's network security configuration subsystem, to make it easier for you to work with SSL-enabled Web sites. This is covered in greater detail in the chapter on SSL.

cwac-presentation

The cwac-presentation repository contains code in support of the `Presentation` system, for sending alternative content to an external display, independent of the device's primary screen.

All of the classes in this repository are covered in the chapter on the `Presentation` system.

cwac-provider

The cwac-provider repository contains `StreamProvider`, a riff on Google's `FileProvider`, offering a "canned" implementation of a `ContentProvider` that can serve files from a variety of sources, such as assets and raw resources from your project.

This is discussed briefly in the chapter on `ContentProvider` implementations.

cwac-sacklist

The cwac-sacklist repository contains `SackOfViewsAdapter`, which implements the `ListAdapter` interface for a collection of individual `Views` that serve as rows. This is used in support of the `MergeAdapter`, for example.

cwac-saferoom

The cwac-saferoom repository contains code that bridges the gap between SQLCipher for Android and the Room database layer from the Architecture

Components. This library is covered in depth in "Android's Architecture Components".

cwac-security

The cwac-security repository contains code to help app developers help their users defend against attacks. At the moment, this contains the `PermissionUtils` class, used to help determine if a custom permission was defined by another app before yours was installed. This is discussed in the chapter on advanced permission techniques.

cwac-strictmodeex

The cwac-strictmodeex repository contains classes that serve a similar role to Android's `StrictMode`, yelling at you for problematic code.

Specifically, this repository contains `StrictAdapter`, which measures the timing of methods like `getView()`, logging information about slow adapters that may result in sluggish `ListView` scrolling.

cwac-wakeful

The cwac-wakeful repository contains the `WakefulIntentService` and related support classes, for doing work and keeping the device awake while that work is going on. This is covered in the chapter on `AlarmManager`.

Appendix B: Android 8.0

In 2017, Google released Android 8.0, code-named "Oreo", followed shortly by Android 8.1. Android 8.1 changed very little in the Android SDK for most developers, but Android 8.0 had some significant changes.

This appendix outlines those changes. In some cases, it serves as pointers to where this material is covered elsewhere in the book. For smaller topics, details of the change appear directly in the appendix.

The War on Background Processing, Continued

Starting with Android 6.0, Google has been trying to limit the impacts of background processing on the device, particularly with respect to battery usage and RAM consumption. Since most background work tends to be invisible to the user, users therefore will tend to blame Android for problems that stem from the users' chosen apps as much, if not more than, from Android itself. As a result, in Android 6.0, Doze mode and app standby were added to curtail periodic work, and Android 7.0 started putting limitations on some types of system broadcasts.

Android 8.0 is furthering Google's objectives in this area, eliminating significant types of background processing.

Background Service Limitations

For apps that have a `targetSdkVersion` over 25 and are running on Android 8.0, background services are limited. After a short period of time — as low as one minute — any such services will be stopped and you will be unable to start new ones.

Also, even if your `targetSdkVersion` is 25 or lower, you might *still* have these limitations applied to your app. If your app appears on the Battery screen in Settings — indicating that it is using above-average power — the user will have the ability to apply these limitations to your app from there.

This is explored in greater detail in the chapter on services.

WakeLock Limitations

If your service holds a `WakeLock`, and that `WakeLock` is not released when the service is stopped, Android will forcibly release the `WakeLock`.

Leaking an acquired `WakeLock` was a bad practice, and since your process can be terminated quickly at any point once you no longer have a running service, developers should have been assuming all along that a `WakeLock` should be released when a service is stopped. Android 8.0 is merely being a bit more aggressive about dealing with these leaks.

Manifest-Registered Broadcast Limitations

For apps that have a `targetSdkVersion` over 25, another limitation comes into play: you cannot receive implicit broadcasts via a manifest-registered receiver.

In other words, if you have a receiver in the manifest that has an `<intent-filter>`, there is a very good chance that it will no longer receive broadcasts.

What Is Affected

Implicit broadcasts are broadcasts using an implicit `Intent`, one that just has an action string (and possibly a `Uri`, categories, or MIME type), but does not identify a specific `BroadcastReceiver`. Explicit broadcasts use an explicit `Intent`, one that does identify a specific `BroadcastReceiver`.

The Android 8.0 limitation affects:

- Implicit broadcasts sent by the system, except for a handful of whitelisted ones
- Implicit broadcasts sent by apps, intended to be delivered to other apps
- Implicit broadcasts sent by apps, intended to be delivered only to themselves, based off of old programming patterns that used system

broadcasts instead of an in-process event bus (e.g., `LocalBroadcastManager`, greenrobot's EventBus)

If your `targetSdkVersion` is 25 or lower, though, your app will not be affected. Also, if you happen to be the one sending the broadcast, and you are requiring a signature-level permission for that broadcast, your app will not be affected, apparently.

Also note that various `Intent` actions documented on the `Intent` class are actually used with explicit broadcasts, not implicit ones. For example, the `ACTION_PACKAGE_REPLACED` broadcast is an implicit one, but `ACTION_MY_PACKAGE_REPLACED` is an explicit one, as that one is only sent to the app that was just upgraded.

The `Intents/PackageLogger` sample project is a very simple app, dominated by an `OnPackageChangeReceiver` that registers for a few `Intent` actions in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.sysevents.pkg"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:allowBackup="false">
    <receiver android:name=".OnPackageChangeReceiver">
      <intent-filter>
        <action android:name="android.intent.action.PACKAGE_ADDED" />
        <action android:name="android.intent.action.PACKAGE_REPLACED" />
        <action android:name="android.intent.action.PACKAGE_REMOVED" />

        <data android:scheme="package" />
      </intent-filter>
    </receiver>

    <activity
      android:name="BootstrapActivity"
      android:theme="@android:style/Theme.Translucent.NoTitleBar">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
```

```
    </activity>
  </application>

</manifest>
```

(from Intents/PackageLogger/app/src/main/AndroidManifest.xml)

...and logs their occurrences to Logcat:

```
package com.commonsware.android.sysevents.pkg;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OnPackageChangeReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Log.d(getClass().getSimpleName(),
      intent.getAction()+" for "+intent.getData().toString());
  }
}
```

(from Intents/PackageLogger/app/src/main/java/com/commonsware/android/sysevents/pkg/OnPackageChangeReceiver.java)

On Android 7.1 and lower devices, this app dutifully logs those events (e.g., when the user installs an app). On Android 8.0, instead, the receiver does not get control, and the following message is recorded to Logcat:

```
W/BroadcastQueue: Background execution not allowed: receiving Intent {
act=android.intent.action.PACKAGE_REMOVED dat=package:com.commonsware.cwac.cam2.demo
flg=0x4000010 (has extras) } to
com.commonsware.android.sysevents.pkg/.OnPackageChangeReceiver
```

Why This Ban Was Added

You might think that the concern was tied to the battery, as this seems like another front in the ongoing "war on background processing" that has been going on since Doze mode was introduced in Android 6.0.

As it turns out, battery is of secondary importance. The real reason is process churn.

Quoting a Google engineer:

> To help understand what is going on, I need to clarify that the purpose of this change is *not* directly related to battery use, but rather to address long-standing issues we have had in the platform where devices that are under memory pressure can get in to bad thrashing states. Very often these states are due to broadcasts: some broadcast or broadcasts are being sent relatively frequently, which a lot of applications are listening to through their manifest (so need to be launched to receive it), but there is not enough RAM to keep all of those app proceses *[sic]* in cache, so the system ends up continually thrashing through processes each time the broadcast is sent.
>
> This is an issue regardless of whether the device is currently plugged in to power. In fact, this can more frequently be an issue on Android TV devices (which are always plugged in to power) because they tend to be fairly tight on RAM!

Workarounds for Senders

If you are using broadcasts for communicating between app components within a single process, switch to using `LocalBroadcastManager`.

If you are using broadcasts for communicating between app components within multiple processes of your own, switch to using explicit broadcasts.

Beyond that, if you are sending implicit broadcasts, you can break through the ban by finding the receivers and sending individual explicit broadcasts instead.

This, and the overall ban, is illustrated in the `Intents/Fanout` sample project. As with some of the event bus samples, this app has a UI that consists of a transcript-mode `ListView`, to which we will append events as they arrive. In this case, the events are broadcasts that we are sending, using different approaches for sending them based on an overflow menu item.

If the user taps the "Explicit" overflow menu item, we create an explicit `Intent` identifying our `TestReceiver` and send that using `sendBroadcast()`. This works, even for an app like this one that has `targetSdkVersion 'O'`, and the broadcast shows up in the list.

If the user taps the "Implicit" overflow menu item, we create an implicit `Intent` tied to the action string used by the `<intent-filter>` of the `TestReceiver`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.broadcast.fanout"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <receiver android:name=".TestReceiver">
      <intent-filter>
        <action android:name="${applicationId}.TEST" />
      </intent-filter>
    </receiver>
  </application>

</manifest>
```

(from Intents/Fanout/app/src/main/AndroidManifest.xml)

However, sending that implicit `Intent` fails, with this warning message showing up in Logcat:

```
W/BroadcastQueue: Background execution not allowed: receiving Intent {
act=com.commonsware.android.broadcast.fanout.TEST flg=0x10 (has extras) } to
com.commonsware.android.broadcast.fanout/.TestReceiver
```

If the user taps the “Fanout” overflow menu item, we create the same implicit `Intent` as before (though we tuck an extra onto it to identify it as the “fanout” case instead of the regular implicit case). And this time, it works. The reason why it works is that rather than sending one implicit broadcast, we send N explicit broadcasts, one for each registered receiver:

```
private static void sendImplicitBroadcast(Context ctxt, Intent i) {
  PackageManager pm=ctxt.getPackageManager();
  List<ResolveInfo> matches=pm.queryBroadcastReceivers(i, 0);

  for (ResolveInfo resolveInfo : matches) {
    Intent explicit=new Intent(i);
    ComponentName cn=
      new ComponentName(resolveInfo.activityInfo.applicationInfo.packageName,
        resolveInfo.activityInfo.name);

    explicit.setComponent(cn);
    ctxt.sendBroadcast(explicit);
  }
}
```

(from Intents/Fanout/app/src/main/java/com/commonsware/android/broadcast/fanout/MainActivity.java)

Unfortunately, this brings back the process churn, and if lots of developers do this, there may be reprisals from Google. You might try introducing some delay between the broadcasts, inside the loop, to spread out the impact. However, this starts to get tricky if you spread it out over more than a few seconds (e.g., do you now need an `IntentService` and a `WakeLock`? what if your process is terminated before the broadcast loop is completed?).

Google recommends that you have the user agree to which of these components should receive the broadcast, perhaps through some sort of `MultiSelectListPreference`. Then, instead of broadcasting to all that match your implicit broadcast, you only broadcast to those that the user has chosen. How practical this is will depend on the app and the desired user experience.

Workarounds for Receivers

If you are receiving system-sent implicit broadcasts (e.g., `ACTION_PACKAGE_ADDED`), keep your `targetSdkVersion` at 25 or lower, until we figure out better workarounds that (hopefully) do not involve polling.

If you are receiving implicit broadcasts from another app, ask the developer of that app what the plan is for Android 8.0. Perhaps they will use the above technique, or perhaps they will switch to some alternative communications pattern.

Background Location Limitations

Background apps — principally, those that do not have the foreground UI and are not a foreground service — will receive fewer location updates than before, whether using `LocationManager` or the Play Services fused location API. The documentation says that background apps will receive location information "only a few times each hour".

Note that this affects *all apps*, not just those with a `targetSdkVersion` over 25.

Besides putting your app in the foreground, you can:

- Use the Play Services geofencing API (detecting when the device enters or leaves a designated area), as these events may not be affected by the limit
- Use the "passive provider" with `LocationManager` (basically asking to get a copy of locations sent to other apps but not otherwise attempting to get locations), as if other apps happen to be in the foreground and requesting locations, you may get those updates as well, despite being in the background

JobScheduler Enhancements

A minor improvement to `JobScheduler` comes in the form of new constraint methods on `JobInfo.Builder`: `setRequiresBatteryNotLow()` and `setRequiresStorageNotLow()`. If you use these, your jobs will not run when the battery is low or when storage is low, respectively.

However, the bigger change really comes in the roles that `JobScheduler` plays:

- The original role was for classic periodic work. You would request to get control every so often (in addition to constraints like the ones cited above). When you got control, you would do some chunk of work, then indicate that the job was finished.
- The role added in Android 7.0: monitoring content for changes. You could add a "trigger content `Uri`" to a `JobInfo.Builder`, indicating that rather than getting control every so often, you want to get control when the content at that `Uri` changes. The net result is the equivalent of a `ContentObserver`, without your app having to have a process running all of the time.
- One role added in Android 8.0 is a work queue. The idea here is that your periodic job is spending a period of time processing any enqueued work,

where the job ends either when there is no more work to be done or when your allotment of time for the job elapses.
- That role is in support of yet another role, as a replacement for `IntentService` for performing background work, by means of a `JobIntentService` offered in the Android Support Library.

`JobIntentService`

`IntentService` still works on Android 8.0, but unless you make it be a foreground service, you will be limited to ~1 minute of runtime before your service is stopped abruptly. `JobIntentService` is a wrapper around a `JobService` that offers `IntentService`-style semantics. On Android 8.0 and higher, when you tell a `JobIntentService` to do some work, it enqueues that work via `JobScheduler`. On Android 7.1 and earlier, the `JobIntentService` behaves more like a regular `IntentService`, though one that supplies a `WakeLock` for you (akin to the author's `WakefulIntentService`).

`JobIntentService` is examined in greater detail in the chapter on services.

JobScheduler as Work Queue

Under the covers, `JobIntentService` is taking advantage of a new capability in `JobScheduler` on Android 8.0: a work queue. `IntentService` had such a queue, after a fashion, in that it would process one `Intent` at a time through `onHandleIntent()`, queuing up other `Intent` objects that arrive while `onHandleIntent()` is busy. `JobScheduler` now offers a similar capability for your `JobService`, where you can post jobs and have your `JobService` end when the work is completed.

This is covered in detail in the chapter on `JobScheduler`.

Auto-Fill

Many Web browsers, such as Chrome, offer "auto-fill". The browser remembers things that you have typed into certain fields, like addresses, and offers to fill those back in when you go to fill in the same form again... or even a different form with similar fields.

Android 8.0 adds auto-fill capability to Android. However, as with the Assist API, Android itself is merely a conduit for form information. The actual implementations of auto-fill, in terms of storing and restoring values, is handled by an auto-fill

service, which the user can enable in Settings. Presumably, when Android 8.0 ships in final form, a Google-supplied auto-fill service will come pre-installed, but users can switch to alternative implementations.

While the documentation suggests that no code changes are required to participate in auto-fill, that is not strictly accurate. Auto-fill relies at least in part upon widget IDs to determine the roles of the widgets, and as a result you may need to consider revising your widget IDs to follow some conventions. Also, you will need to consider whether some or all of your activities really should be participating in auto-fill, for privacy or security reasons.

Auto-fill is covered in detail in a separate chapter.

Notification Channels

Through Android 7.1, all notifications were created equal. In other words, the user could apply a single set of rules for configuring how those notifications were delivered from your app. How *much* configuration was even available varied by Android OS version, but on newer devices users could control whether to block those notifications, show them without any sound or vibration, what amount of information should be shown on the lockscreen, and so on.

In Android 8.0, all notifications go into channels that you initially define. The user then has the ability to control the behavior of notifications on a per-channel basis. This offers finer granularity of control over the breadth of notifications that your app might deliver. Android 8.0 also extends the depth of control, with more options for the user to decide how a channel's notifications should work.

These are covered briefly in the chapter on notifications and in greater detail in the chapter on advanced `Notification` techniques.

Other Changes with Notifications

Android 8.0 offers a few additional features of note for notifications, features that are independent of — or only partially tied into — the new notification channels: auto-timeout, colorized notifications, and launcher icon badges. These are covered in the chapter on advanced `Notification` techniques.

Multi-Window Changes

Android 7.0 introduced a native implementation of multi-window, with a particular emphasis on split-screen presentations on phones and tablets.

Android 8.0 extends this, particularly adding picture-in-picture support for mobile devices, and extending multi-window to also work on external displays, such as TVs and monitors. These changes are covered in the chapter on multi-window support.

WebView Changes

`WebView`, as Android developers use it, is really a facade API, as of Android 5.0. Previously, the `WebView` implementation was part of the Android framework. Nowadays, the implementation is delegated to the Android System WebView, which allows Google to update the `WebView` implementation without relying upon manufacturers to distribute firmware updates.

Hence, to some extent, `WebView` continuously changes, as the Android System WebView app gets updated a few times per year.

However, it is only in a new version of Android that Google changes the API or the general approach taken by a `WebView` implementation. In Android 8.0, two significant changes were made: multi-process mode and support for banning of cleartext traffic. These are covered in the chapter on advanced `WebView` usage.

ContentProvider Changes

`ContentProvider` got a few new features in Android 8.0 as well, related to paged queries and requesting data refreshes. Unfortunately, these changes are completely undocumented.

These changes will be examined in a future edition of this book.

Storage Access Framework Changes

The Storage Access Framework underlies `ACTION_OPEN_DOCUMENT`, `ACTION_CREATE_DOCUMENT`, and related actions, providing you with the equivalent of the “file open” and “save as” dialogs you may be used to in other programming

environments. This framework received a few extensions in Android 8.0, which are outlined in the chapter on consuming documents.

Package Management

Apps that install or delete other apps will experience some changes in Android 8.0, as well apps that have used broadcasts like `ACTION_PACKAGE_ADDED` to find out about changes in the mix of installed apps. These changes are covered in detail in the chapter on miscellaneous integration techniques.

Fonts as Resources

While Android has long had a `Typeface` class, using it was a pain. There was no way in layout resources to reference a custom `Typeface`, and so you had to apply the `Typeface` directly in Java code. At best, you could use a library like Calligraphy to try to simplify this.

Android 8.0, for the first time, makes fonts a first-class type of resource.

You can put OTF and TTF fonts into `res/font/` directories, or in other directories based upon resource set qualifiers (e.g., `res/font-v26/`). The `font` resource directories can also hold `<font-family>` XML files that tie multiple OTF/TTF fonts together into a single font family, for cases where different styles or weights are stored as separate font files.

Then, in layouts and style resources, you can use `android:fontFamily` on `TextView` and its subclasses, to indicate that the `TextView` should use fonts from your chosen family. Then, attributes like `android:textStyle` will pull fonts from the font family, and those fonts will be applied to the text in the `TextView`.

Also:

- The `Resources` object (obtained via `getResources()` on a suitable `Context`) has a `getFont()` method to retrieve a `Typeface` for a given font resource, given its `R.font` resource ID (e.g., `R.font.hack`)
- Information about system-supplied fonts can be obtained via a `FontManager` system service and its `getSystemFonts()` method

Other Major Changes in Android 8.0

While Android 8.0 does not have very many user-facing changes, it does have quite a few that affect developers, beyond those already listed.

Cache Quotas

Historically, `getCacheDir()` and `getExternalCacheDir()` represented "cache" directories. There were no limits as to what we could store there, in terms of available space. However, we had to recognize that Android — and select third-party apps — might clear our cache at any point.

Android 8.0 introduces a "cache quota". The name brings to mind a hard quota, where we crash or something if we exceed the limit. In truth, this appears to be a soft quota: apps exceeding the quota are likely to have their caches cleared, if and when Android needs to free up disk space.

The `StorageManager` system service has two methods related to this:

- `getCacheQuotaBytes()` tells you what your quota is
- `getCacheSizeBytes()` tells you how much cache space you are using

Both methods take a `File` object and return the corresponding values for cache stored on the filesystem on which that `File` is stored. For most Android devices, internal and external storage are on the same partition with the same filesystem, but that is not the case for *every* device. Hence, you will want to check both internal and external storage, using appropriate `File` objects, if you are storing data in both places.

You might use these methods as part of your own manual cache trim operation. For example, you might set up an idle-time once-a-day job with `JobScheduler` to examine what is in cache and delete a few things to get you under the cache quota. This would reduce the likelihood that Android will wipe out all of your cache, because you exceeded the quota and Android needed to free up the disk space for the user.

Content Annotations

Usually, when we want a chooser for some activity `Intent` that we want to start, we use `Intent.createChooser()` to wrap our `Intent` in another `Intent`. That "outer"

`Intent` will be an `ACTION_CHOOSER` `Intent`, and that `Intent` can itself be configured via extras.

One new `ACTION_CHOOSER` extra is `EXTRA_CONTENT_ANNOTATIONS`. You can put an `ArrayList` of up to three strings into this extra. Those will be used to help rank the apps that appear in the chooser.

This appears to be based on a learning algorithm; activities that are among the candidates do not somehow advertise what "content annotations" they are good at. Rather, the idea appears to be that Android will learn over time which apps the user wants to use for which types of content, based on these annotations.

The documentation for `EXTRA_CONTENT_ANNOTATIONS` provides a list of nouns to choose from for these annotations. Exactly how your app is supposed to determine which of those nouns is relevant for any particular chooser is unclear.

Seekable Streams

A major headache with the migration of Android away from sharing files to sharing streams from `ContentProviders` is the issue of seekability. Many apps want to be able to use methods like `mark()` and `reset()` on an `InputStream` as they are processing content. However, that implies a "seekable" stream, one where we can rewind and re-read bytes seen (or skipped) previously. In general:

- Streams created directly from files (e.g., `ParcelFileDescriptor.openFile()`) were seekable
- Streams created via pipes (e.g. `ParcelFileDescriptor.createPipe()`) were not seekable

For content that already exists as an ordinary file, this was not much of a limitation, as using file-based streams was already the easy solution. However, not all content is available as ordinary files, such as:

- Content stored as files, but in an encrypted or otherwise encoded format that needs to be decoded as part of serving it to other apps
- Content stored in `BLOB` columns of a SQLite database
- Content that is in "the cloud" and has not yet been downloaded to the device

The net effect was that many apps, for maximum compatibility with *other* apps, would need to create a file with all of the content, wasting disk space and possibly harming privacy and security.

In Android 8.0, `StorageManager` offers support for "proxy file descriptors", via methods like `openProxyFileDescriptor()`. These give you a `ParcelFileDescriptor`, for use in methods like `openFile()` on a `ContentProvider`. However, rather than assuming a particular file or a particular pipe, a `ProxyFileDescriptorCallback` is used to ask you to provide access to the data, based on offsets and sizes. How you implement that is up to you. For example, for content backed by a `BLOB` column, you might cache the content as a byte array, then return segments of that byte array as needed by the `ProxyFileDescriptorCallback`.

However, `ProxyFileDescriptorCallback` leaves *much* to be desired, to the point where it is useless for most clients and providers.

Back in the chapter on `ContentProvider` patterns, we had a demo of using a `ParcelFileDescriptor` pipe, via `createPipe()`, as a way of returning custom data from a `ContentProvider`. The `ContentProvider/ProxyPipe` sample project is an updated version of the original `Pipe` example, adjusted to build using Android 8.0 and to use `ProxyFileDescriptorCallback`.

Implementing a ProxyFileDescriptorCallback

Our `ProxyFileDescriptorCallback` implementation is called `BufferProxyCallback`, so named as it will stream back the contents of an in-memory buffer. This is not particularly realistic, but it does simplify the example a fair bit, focusing on the `ProxyFileDescriptorCallback` API:

```
package com.commonsware.android.cp.pipe;

import android.os.ProxyFileDescriptorCallback;
import android.system.ErrnoException;
import android.system.OsConstants;
import hugo.weaving.DebugLog;

class BufferProxyCallback extends ProxyFileDescriptorCallback {
  private final byte[] buffer;

  @DebugLog
  BufferProxyCallback(byte[] buffer) {
    this.buffer=buffer;
  }

  @DebugLog
  @Override
  public void onRelease() {
    // not needed here
  }

  @DebugLog
  @Override
```

```
  public long onGetSize() throws ErrnoException {
    return(buffer.length);
  }

  @DebugLog
  @Override
  public int onRead(long offset, int size, byte[] data) throws ErrnoException {
    int toRead=(offset+size<=buffer.length) ? size : (int)(buffer.length-offset);

    System.arraycopy(buffer, (int)offset, data, 0, toRead);

    return(toRead);
  }

  @DebugLog
  @Override
  public int onWrite(long offset, int size, byte[] data) throws ErrnoException {
    throw new ErrnoException("onWrite", OsConstants.EOPNOTSUPP);
  }

  @DebugLog
  @Override
  public void onFsync() throws ErrnoException {
    // not needed here
  }
}
```

(from ContentProvider/ProxyPipe/app/src/main/java/com/commonsware/android/cp/pipe/BufferProxyCallback.java)

`ProxyFileDescriptorCallback` is an `abstract` class, and there are five methods that you need to override to fulfill the contract:

- `onGetSize()`, which returns the number of bytes in the content, which in this case is the length of the buffer.
- `onRead()`, which needs to copy a specified sequence of bytes, based on an offset from the front of the content and a length, to a supplied byte array. Here, `onRead()` uses `System.arraycopy()` to copy from buffer to buffer, either to the requested length (`size`) or the remaining bytes (the length of the buffer, minus the offset). It needs to return the number of bytes that are read.
- `onWrite()`, which needs to take the bytes from a supplied byte array and update a region of the content to match. Here, we are implementing a read-only proxy, and so we raise an `ErrnoException`, indicating that this particular operation is not supported. If this method were implemented "for real", `onWrite()` needs to return the number of bytes actually written.
- `onFsync()` needs to ensure that any writes created by `onWrite()` are fully committed to disk (or the equivalent if disk is not the backing store for the data). Here, there is nothing to "sync", so this method is ignored.
- `onRelease()`, which is called when this particular proxy is no longer needed, so you can release any resources that you might be holding.

`onRead()` is particularly tricky to implement for things other than an in-memory buffer. The documentation states that "It needs to return exact requested size of bytes unless it reaches file end". Many I/O options do not have a read operation that guarantee an exact number of bytes to be read. As a result, the proxy may need to do some caching of bytes from a previous read operation, to use on the next one, when the number of requested bytes does not match the number of bytes that were read off of whatever I/O channel is being used.

These methods all are capable of throwing an `ErrnoException`. That is a rather new and obscure exception, and it actually points to a problem that most developers will encounter with `ProxyFileDescriptorCallback`... which we will see in a bit.

Using a ProxyFileDescriptorCallback

A `ContentProvider` can then use a `ProxyFileDescriptorCallback` in concert with `StorageManager` and `openProxyFileDescriptor()`, in places like `openFile()`:

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
  throws FileNotFoundException {
  AssetManager assets=getContext().getAssets();

  try {
    InputStream in=
      assets.open(uri.getLastPathSegment(), AssetManager.ACCESS_STREAMING);
    byte[] content=readAll(in);

    StorageManager sm=getContext().getSystemService(StorageManager.class);

    return(sm.openProxyFileDescriptor(ParcelFileDescriptor.MODE_READ_ONLY,
      new BufferProxyCallback(content)));
  }
  catch (IOException e) {
    Log.e(getClass().getSimpleName(), "Exception opening pipe", e);

    throw new FileNotFoundException("Could not open pipe for: "
      +uri.toString());
  }
}
```

(from ContentProvider/ProxyPipe/app/src/main/java/com/commonsware/android/cp/pipe/PipeProvider.java)

Here, we first open an `InputStream` on a PDF file stored in `assets/`, then pass that stream to a `readAll()` method that reads in the entire PDF into a buffer:

```
// inspired by http://stackoverflow.com/a/17861016/115145

public static byte[] readAll(InputStream is) throws IOException {
  try (ByteArrayOutputStream baos=new ByteArrayOutputStream()) {
    byte[] buf=new byte[16384];

    for (int len; (len = is.read(buf)) != -1; ) {
      baos.write(buf, 0, len);
    }

    baos.flush();
    is.close();

    return(baos.toByteArray());
  }
}
```

(from ContentProvider/ProxyPipe/app/src/main/java/com/commonsware/android/cp/pipe/PipeProvider.java)

Then, we obtain a `StorageManager` via `getSystemService()` and call `openProxyFileDescriptor()`, passing in the read/write mode that we want (in this case, hard-wiring it to `MODE_READ_ONLY`) and the `ProxyFileDescriptorCallback` that we want to use (here, a `BufferProxyCallback` wrapped around that buffer).

The Woes of ProxyFileDescriptorCallback

`ProxyFileDescriptorCallback` is intrinsically incompatible with Java I/O. Java I/O classes and methods are a (relatively) high-level interface to the low-level sorts of functions that C/C++ developers use, like `fread()`. As a result, it is unlikely that you will find `ProxyFileDescriptorCallback` to be easy to use.

Also, the results add no real value over just using a pipe... unless the client jettisons all Java I/O (including anything layered on top of it) and uses that same low-level I/O API.

Foreground Heap Compaction

When Android first shipped, the runtime environment that ran our apps — Dalvik – offered no heap compaction in its garbage collection. As a result, our heap would become fragmented, looking like a block of Swiss cheese, with lots of smaller free blocks of memory. This was in contrast to the way garbage collection worked on the Java VM, where the heap was compacted regularly. As a result, we would get an `OutOfMemoryError` very frequently. We were not out of heap space, but there was no

single block of memory big enough for whatever we were trying to allocate. All we had were lots of smaller blocks.

Android 5.0 introduced ART as a new runtime environment. ART's garbage collector *would* compact the heap, moving objects around in memory to coalesce all those smaller blocks of free memory into one large block. However, it would only compact the heap when the app was in the background, as otherwise the heap compaction work might slow the app down too much at points in time when the user might notice.

Android 8.0's update to ART brings a new garbage collection algorithm, one that attempts to compact the heap while our app is in the foreground, as well as when it is in the background. This should further reduce spurious `OutOfMemoryErrors`, limiting those to cases when we are truly running out of heap space or are trying to allocate some ridiculously large block.

Shortcuts, App Widgets, and Pinning

Home screen implementations in Android have long had support for shortcuts. However, from an Android SDK standpoint, shortcuts did not exist, by and large. You could use `ACTION_CREATE_SHORTCUT` to allow the user and an app to define the `Intent` to be used as a shortcut... and that was about it. Some apps relied upon an undocumented, unsupported `com.android.launcher.action.INSTALL_SHORTCUT` means of asking a home screen to create a shortcut, though home screen implementations were never required to support it.

Android 7.1 added a more formal definition of shortcuts, initially aimed at providing them as options when the user long-pressed on a launcher icon in the home screen. This also introduced `ShortcutManager` as a way for an app to define available shortcuts dynamically.

Android 8.0 steers developers away from `com.android.launcher.action.INSTALL_SHORTCUT` and to `requestPinShortcut()` on `ShortcutManager` as a way to ask the launcher (and the user) to set up a shortcut.

Android 8.0 also offers a similar way to prompt the user to set up an app widget. For launchers that have support for this (reported by `isRequestPinAppWidgetSupported()` on `AppWidgetManager`), you can call `requestPinAppWidget()` on `AppWidgetManager` to try to convince the user to set up an app widget.

These capabilities will be examined in greater detail in an upcoming edition of this book.

Auto-Sizing TextView

Android 8.0 added the ability to automatically change the font size in a `TextView` based upon the length of the text, something that previously we obtained from third-party libraries. This is covered in the chapter on fonts and text.

SMS Tokens

In Android 4.4, Google significantly changed the behavior of SMS messages. In particular, all messages would wind up in the user's SMS client. Previously, it was possible for apps to monitor and intercept messages, and some developers used this for a communications channel. For example, to validate a user's phone number, the server might send an SMS message to that number, which the app would intercept and process, bypassing the user's SMS client. On Android 4.4, that flow did not work as well, as the message would still wind up in the SMS client, whether the app responsible for the message wanted that or not.

Android 8.0 provides "app-specific SMS tokens", via a `createAppSpecificSmsToken()` method on `SmsManager` as a way of accomplishing this flow. This is covered in detail in the chapter on SMS.

Custom Preference Storage

Android 8.0 adds `PreferenceDataStore`. In theory, this allows for pluggable ways to persist `SharedPreferences`, such as storing them in an encrypted container. In practice, it does not work especially well. The details of `PreferenceDataStore` – and its problems — are found in the chapter on advanced preferences.

FragmentLifecycleCallbacks

In API Level 14, we were given `ActivityLifecycleCallbacks`. We can register an instance of this interface on an `Application` object, and then receive callbacks for every lifecycle method for every one of our app's activities. This allows us to implement cross-cutting concerns (e.g., lifecycle method logging) without having to implement those concerns on each and every activity class.

Android 8.0 gives us `FragmentLifecycleCallbacks`. We can register an instance of this interface with a `FragmentManager`, and then receive callbacks for every lifecycle method for every one of the fragments managed by that `FragmentManager`. This should allow us to implement cross-cutting concerns (e.g., lifecycle method logging) without having to implement those concerns on each and every fragment class.

As with everything involving fragments, there are two implementations: one for the native fragments and one for the fragments backport in the `support-fragment` artifact. This is another situation where you might elect to use the fragments backport, even on an app with a `minSdkVersion` over 11, so you can use this new capability even on older devices.

Other Minor Changes in Android 8.0

And, of course, there are lots of other smaller changes floating around Android 8.0, including those outlined here.

Permission Granularity

With the runtime permission system introduced in Android 6.0, we request *permissions*. However, the UI presented to the user allows them to grant *permission groups*. Presumably, this is to simplify the user experience.

However, in Android 6.0 through 7.1, even though you might request only one permission out of the group, you would actually be granted all the permissions in the group. So, for example, requesting `READ_EXTERNAL_STORAGE` would also allow you to write to external storage, without having specifically requested this via `requestPermissions()`.

This bug is fixed in Android 8.0, for apps with a `targetSdkVersion` over 25. Now, you are only granted the permissions that you request.

This is not supposed to change the user flow, though. Suppose that you call `requestPermissions()` and ask for `READ_EXTERNAL_STORAGE`. The user is prompted whether to give your app the rights associated with the storage permission group. If the user agrees, you get `READ_EXTERNAL_STORAGE`, and only `READ_EXTERNAL_STORAGE`. However, if later you call `requestPermissions()` for `WRITE_EXTERNAL_STORAGE`, in theory, the user will not be prompted regarding this new permission, since the user already granted you all the rights for the storage permission group. Your

`onRequestPermissionsResult()` method should be invoked immediately, indicating that you now have access.

This supposed flow, though, never worked. The user was always prompted for all permissions listed in `requestPermissions()`, even if the permission group had already been granted. This is why various sample apps in this book employ a `netPermissions()` method, so that we only ask for the permissions that we need that we do not already have.

Tooltips

Mouse-centric environments — like classic desktop operating systems — have long had tooltips, where an information bubble appears when you hover over a widget. Android has had the occasional tooltip-like feature, such as the tooltip that you get on action bar items when long-pressing them. Android 8.0 extends this to all widgets. Simply provide the desired text via `android:tooltipText` or `setTooltipText()`. A tooltip will then appear when either the user long-presses the widget or if the user hovers over the widget. In both cases, though, if the long-press or hover event is consumed by something else, the tooltip is not shown.

This is covered in greater detail in the chapter on keyboards and mice.

ZoomButton Deprecated

The ill-used `ZoomButton` widget is now formally deprecated. Most likely, if you are still using it, you should move to something else, such as:

- zoom gestures (e.g., pinch-to-zoom)
- a pair of `ImageButton` widgets

Alternatively, you might consider forking the source code to `ZoomButton` and maintaining your own copy.

App Categories

Apps can use `android:appCategory` in the `<application>` element to suggest a category to use for grouping this app with other similar apps on the device. Unfortunately, there is no documentation for the valid values, and so this attribute is unusable in its current state.

Changes in ANDROID_ID Scope

`ANDROID_ID` has been a long-troubled identifier for Android devices. In the early days, it might be missing or set to some hard-coded value, rather than being a unique identifier. Separate users on a shared Android device get separate `ANDROID_ID` values, so it is not a per-device identifier anyway. On the whole, Google does not recommend the use of `ANDROID_ID`, as seen by its conspicuous absense from their identifier guidelines.

In Android 8.0, `ANDROID_ID` changes yet again, to now be a per-app/per-user identifier. Each app, for each user on a device, gets its own `ANDROID_ID` value. However, Google does guarantee that the value will remain consistent for that app/user combination, surviving even an uninstall and reinstall. However, because the value differs by apps, apps in a suite (or apps sharing a common advertising library) cannot correlate actions, as they will each get their own `ANDROID_ID` value.

For the record, you get the `ANDROID_ID` value via:

```
Settings.Secure.getString(getContext().getContentResolver(),
Settings.Secure.ANDROID_ID);
```

Alert Windows

The "chat heads" offered by Facebook's Android app kicked off a spur of interest in how to create windows that can float over arbitrary apps. This facility was never intended for widespread use, and presumably it adds challenges to future multi-window scenarios.

The technique to create these windows involves using `WindowManager` to create a new window with particular types, like `TYPE_PHONE` or `TYPE_SYSTEM_ALERT`.

On Android 8.0, apps with a `targetSdkVersion` over 25 cannot use these window types anymore. Instead, a new type, called `TYPE_APPLICATION_OVERLAY`, has been added, which apps can use, subject to some limitations:

- These overlay windows do not float over system UI elements, such as the navigation bar or soft keyboards
- Your requested window size and position may be changed by Android as needed "to improve screen presentation"
- The user can block your app from showing these windows

Build.SERIAL Versus Build.getSerial()

One of the values that has appeared on the `Build` class is `SERIAL`, advertised as being a serial number for the device. Whether or not it actually *is* a serial number has been up to device manufacturers.

On Android 8.0, `Build.SERIAL` is deprecated. More importantly, if your `targetSdkVersion` is over 25, `Build.SERIAL` will be set to the string `"unknown"`.

Instead, you need to start using a new `getSerial()` method on the `Build` class. However, this requires the `READ_PHONE_STATE` permission, which, as a `dangerous` permission, requires you to request it at runtime.

StorageStatsManager

`UsageStatsManager` was added in API Level 21, to provide access to historical app usage data.

A similar class, `StorageStatsManager`, is part of Android 8.0, to provide information about storage usage, for things beyond what `StatFs` has provided developers for years. In particular, you can get:

- the "marketing" explanation of the data storage on some volume, which tends to be higher than the actual amount of storage on that volume
- the amount of free space taking into account the amount of used cache that could be freed up by the system
- the storage usage by particular user accounts
- the storage usage by particular apps

As with `UsageStatsManager`, `StorageStatsManager` requires that you request the `PACKAGE_USAGE_STATS` permission. The user cannot grant you this permission directly (e.g., via runtime permissions), but this will cause your app to appear in Settings in a place where the user can grant you this permission.

Also, `setCacheBehaviorAtomic()` allows you to control whether deleting the cache always happens "atomically" (i.e., deleting everything) or whether Android can delete individual cached files. You would specifically want to opt into atomic behavior if you cannot deal with Android deleting some cache files but not others.

TextClassificationManager

If you have ever used Google Translate, and watched it guess what language your pasted-in text was written in, `TextClassificationManager` offers that same "guess the language" feature. Given an instance of the `TextClassificationManager` system service, calling `detectLanguages()` on it will return details of what language(s) were detected, in the form of a `List` of `TextLanguage` objects. From there, you can find confidence scores and the like, to understand how confident Android is in its guess.

Also, given a `TextClassificationManager`, you can obtain a `TextClassifier`. This appears to be an updated edition of `Linkify`: a scanner that can find linkable entities (e.g., email addresses, physical addresses, URLs, phone numbers) and provide information to you about them for the purposes of applying markup.

Appendix C: Community Theater and the Appinars

In addition to the book chapters themselves and the related source code for the sample apps, you also have access to "appinars". Appinars are app-based training, blending video, slides with voiceovers, source code samples, and more.

The APK edition of the book has an embedded appinar player, called Community Theater, along with an embedded roster of available appinars. You can browse through those appinars, download the ones of interest, and play them through Community Theater.

Note that this feature is only available if you are using the APK edition on Android 4.4 or higher.

Viewing the Appinar Roster

In the main book reader, the action bar overflow has a "Community Theater" option:

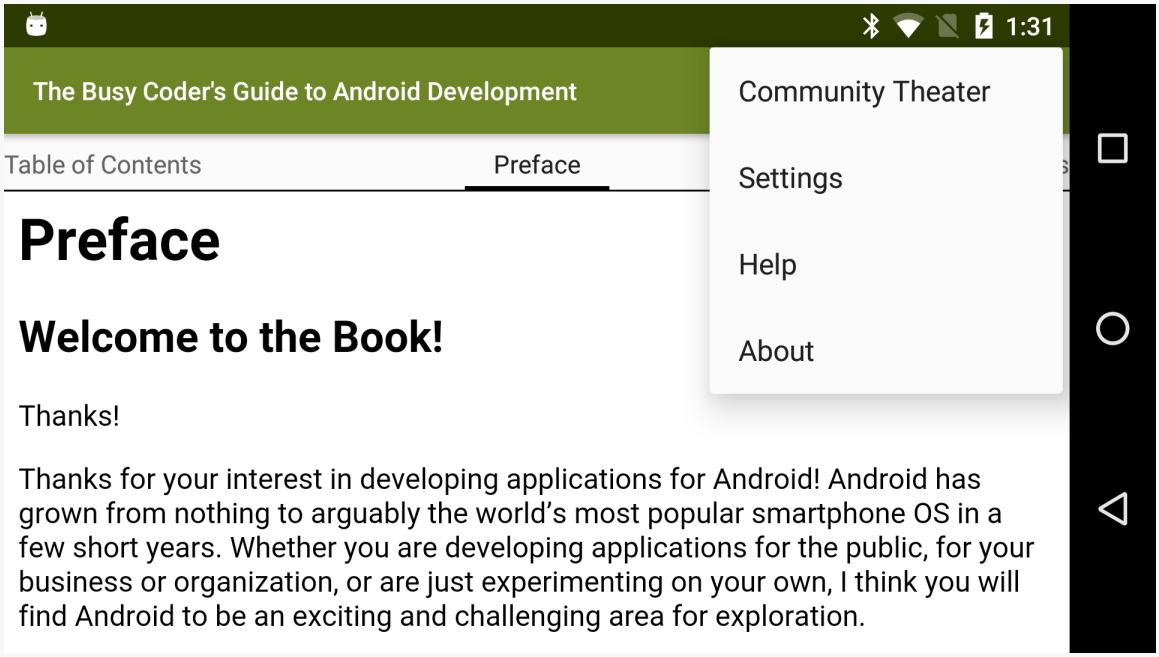


Figure 1137: Book Reader, Showing Overflow

Tapping that will bring up a roster of available appinars, broken down into categories:

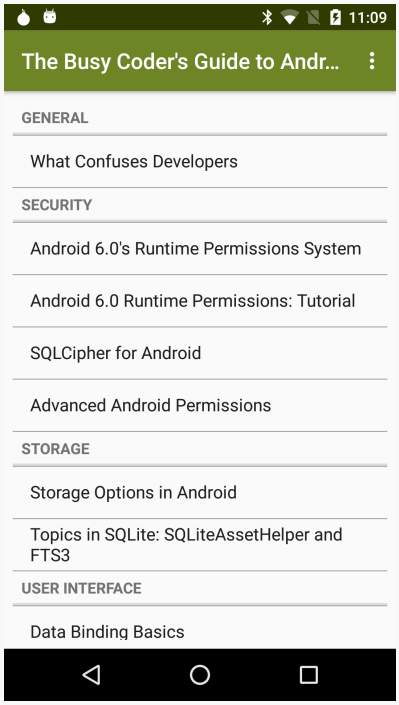


Figure 1138: Appinar Roster, Showing Categories and Appinars

Tapping an appinar brings up details for that appinar:

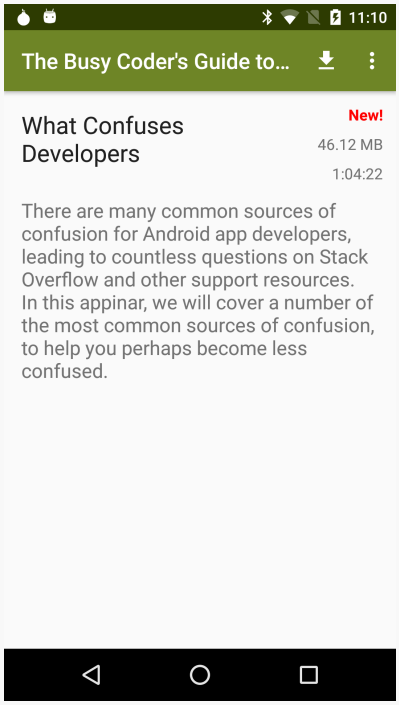


Figure 1139: Appinar Roster, Showing Appinar Details

New appinars will be added with each book update. Existing appinars might be updated to reflect new content or to fix egregious bugs.

Managing Appinars

Tapping the "download" action bar item will download the appinar to your machine. The approximate amount of data to be downloaded, in the form of a ZIP file, is shown above the button.

While the download is going on, an progress bar will be visible in the appinar detail screen, as well as in a notification:

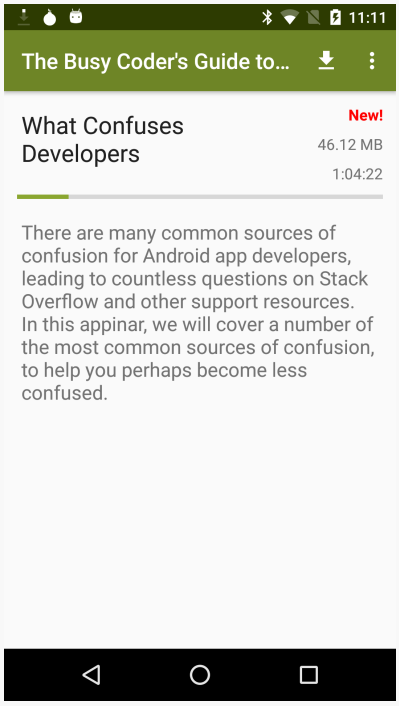


Figure 1140: Appinar Roster, While Appinar is Downloading

Once the appinar is downloaded, the action bar will have options to play or delete the appinar:

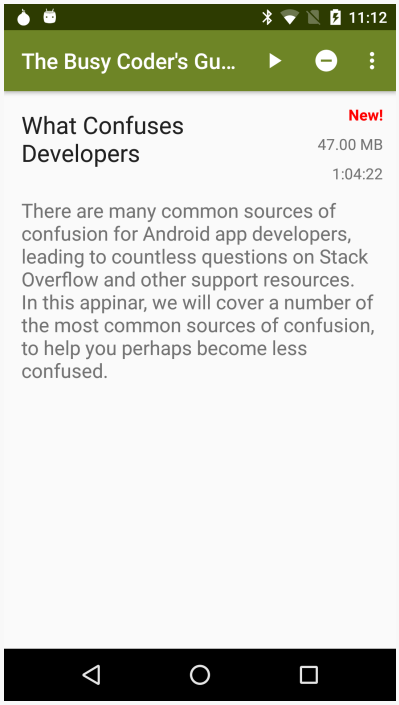


Figure 1141: Appinar Roster, Showing Downloaded Appinar

Tapping the delete action bar item will bring up a confirmation panel:

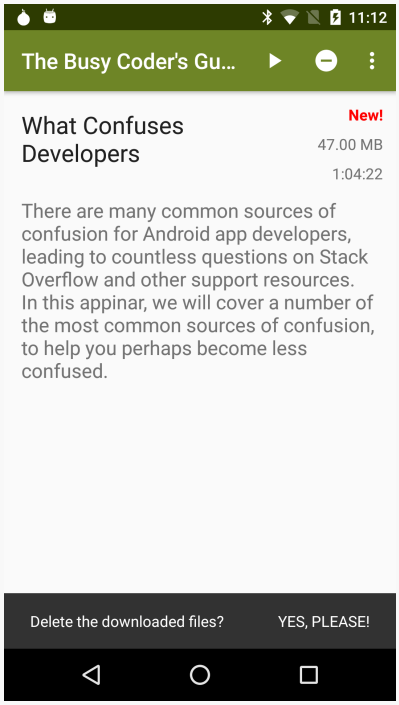


Figure 1142: Appinar Roster, Showing Delete Confirmation

If you confirm the request, the appinar will be deleted.

Viewing an Appinar

Viewing an appinar is merely a matter of downloading it, then tapping the play button in the action bar. Playback should begin immediately, usually with a title slide followed by a brief opening video:

Figure 1143: Player Screen, Showing a Balding Guy

Eventually, when the appinar ends, you will be taken back to the roster of available appinars, where you were when you tapped the play button originally.

Note that while you can view the roster in portrait or landscape mode, playback is locked to landscape, due to the aspect ratio of the videos.

Pausing Playback

If you double-tap the screen while an appinar is playing, playback will be paused. The action bar will appear at the top of the screen, allowing you to navigate through the appinar and so on.

To resume playback, press the BACK button, or double-tap the screen again.

Navigating the Appinar

When playback is paused via a double-tap, and the action bar is visible, you can open a navigation drawer, via the “hamburger” icon in the action bar:

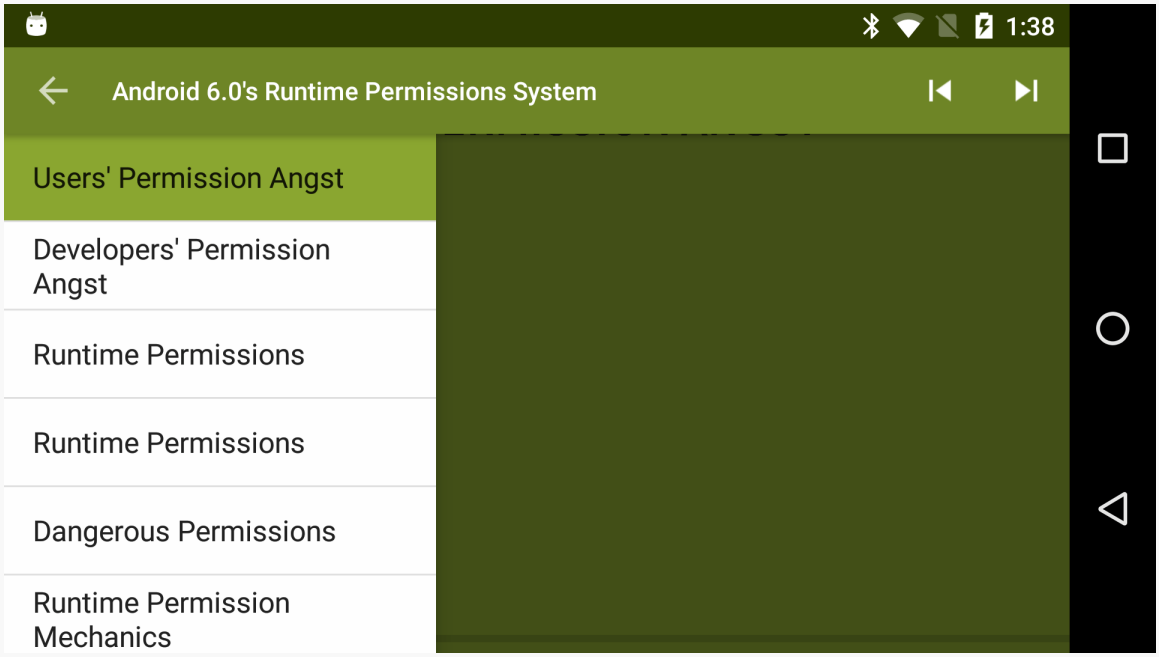


Figure 1144: Player Screen, Paused, Showing the Navigation Drawer

This contains a list of the different "scenes" of the appinar. Tapping on a scene will resume playback, jumping to that particular scene.

The action bar also has "fast-forward" and "rewind" buttons. The fast-forward button jumps to the next scene. The rewind button will start the current scene over from the beginning, if you are well into playback of the scene. If you are close to the start of the scene, the rewind button will take you to the previous scene.

Manipulating the Content

Parts of the appinar may show full source code listings. You can use pinch-zoom gestures to change the font size, if the size is too small for your current display. Pinch-zoom gestures also work on some full-screen images, if the image is significantly larger than your display.

Most of the source code scenes, and a few others noted in the voiceovers, will have "share" action bar items.

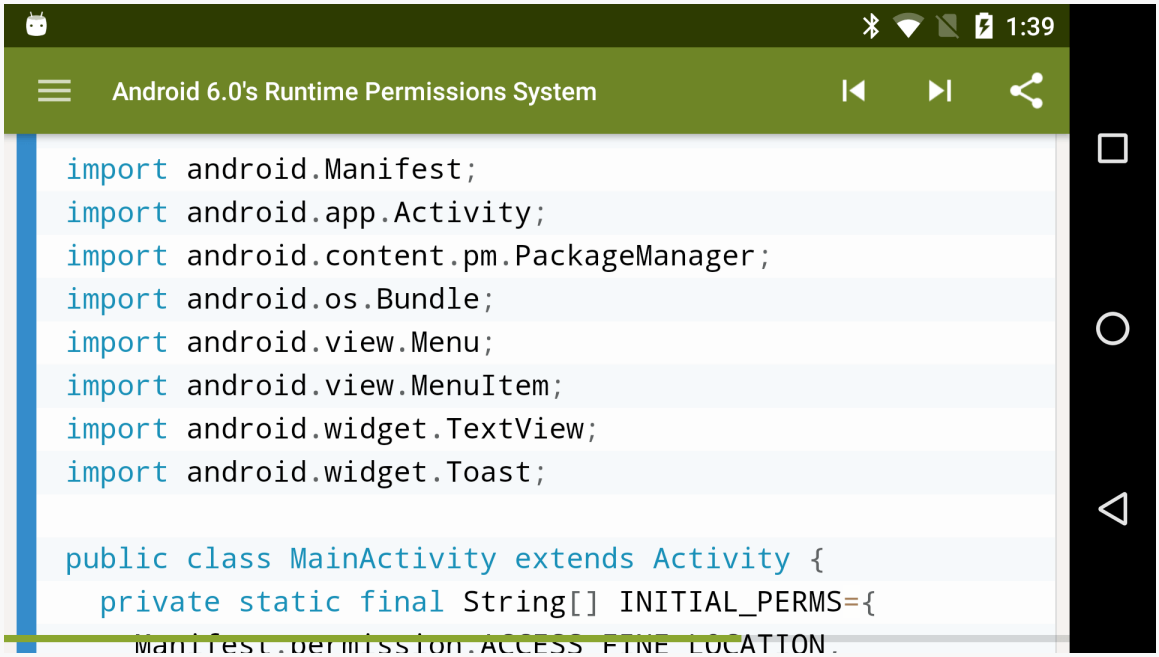


Figure 1145: Player Screen, Paused, Showing Action Bar with Share Item

When playback is paused and the action bar is visible, you can tap that share icon to view the source code (or Web page or other resource) in a separate app, or send the URL to the resource through some other app to another person or Web service (e.g., forward a URL via email or SMS).